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

Progetto e implementazione di regole attive all'interno di ODB-Tools

Relatore:

Chiar.mo Prof. Sonia Bergamaschi

Tesi di Laurea di:

Andrea Salvarani

Correlatore:

Dott. Ing. Maurizio Vincini

Controrelatore

Chiar.mo Prof. Flavio Bonfatti

Anno Accademico 1997 - 98

Parole chiave:

Regole attive
Basi di dati attive
Eventi
ODBMS
ODMG-93

RINGRAZIAMENTI

Ringrazio la Professoressa Sonia Bergamaschi per il prezioso aiuto fornito alla realizzazione della presente tesi, l'Ing. Maurizio Vincini e l'Ing Alberto Corni per l'aiuto nei problemi tecnici.
Inoltre un ringraziamento va all'Ing. Simone Montanari e al laureando Andrea Zaccaria per il sostegno morale durante gli anni di studi.
Infine un grazie alla mia famiglia che ha reso possibile tutto ciò.

Indice

Capitolo 1

Introduzione

Premessa Il lavoro della presente tesi è stato svolto all'interno del gruppo di ricerca sull'applicazione di tecniche di intelligenza artificiale alle basi di dati ad oggetti, presso il Dipartimento di Scienze dell'Ingegneria (DSI) dell'Università di Modena.
Tale gruppo ha in corso di sviluppo il progetto ODB-Tools.
Gli argomenti trattati riguardano basi di dati, modelli e linguaggio di programmazione ad oggetti; si presuppone che il lettore abbia una discreta preparazione su questi argomenti.

Scopo della tesi Estensione di alcuni moduli del progetto ODB-Tools.

1. Estensione del traduttore in modo tale che accetti in ingresso la descrizione di uno schema di database ad oggetti completo delle regole attive, descritto secondo il linguaggio ODL dello *standard* ODMG-93.

2. Estensione dell'OCDL-designer in modo tale che accetti in ingresso lo schema completo di regole attive tradotto dal traduttore.

3. Implementazione delle regole attive in UNISQL tramite un traduttore che sfrutta le strutture dati create dall'OCDL-designer.

Contenuto della tesi Questo documento contiene una panoramica sui database con comportamento attivo e sulle loro funzionalità, una visione di insieme del progetto ODB-Tools e una descrizione dei problemi incontrati e risolti durante la realizzazione del software.
Nel capitolo 2 troviamo una descrizione dettagliata delle funzionalità che deve avere un database attivo, nei capitolo 3 e 4 sono descritti alcuni di questi sistemi soffermandosi sulla sintassi per la definizione delle regole. Nel capitolo

5 si descrive l'ambiente di ODB-Tools preesistente mentre dal capitolo 6 si espone il lavoro svolto per la realizzazione delle estensioni.

Terminologia utilizzata Un DBMS è un software per creare, mantenere, operare su un grande, integrato, multiutente database (DB). Implementa la possibilità di salvare e recuperare dati, controlla la concorrenza, controlla gli accessi e permette l'eventuale recovery. La collezione dei dati, incluse le informazioni secondarie sono salvate nel DB. La struttura del DB è definita dallo schema del database. Lo schema è specificato usando il Data Definition Language (DDL) e l'accesso alle informazioni del DB è dato dal Data Manipulation Language (DML). I due linguaggi rappresentano il così chiamato Data Model (modello dei dati). Infine un sistema database è un DBMS insieme a una collezione di dati (DB).
Un ADBMS (Active Database System Management) estende un DBMS passivo con la possibilità di specificare un comportamento reattivo. Di seguito si introducono i concetti riguardanti la specifica e l'implementazione di queste funzionalità aggiuntive.
Regole ECA (regole Evento - Condizione - Azione) consistono in eventi, condizioni e azioni. Il significato di questo concetto è "Quando avviene un evento, verifica la condizione e se è vera esegui l'azione", una regola attivata si dice "triggered ", un evento scatenante si dice "evento di triggering". Una volta che un insieme di regole è stato definito l'ADBMS monitorizza gli eventi, rilevando l'occorrenza di quelli significativi e notificandola al componente responsabile dell'esecuzione delle regole. Questa notifica è chiamata "signalling" (segnalazione) dell'evento. Conseguentemente, tutte le regole che sono definite per rispondere a questo evento sono triggered e devono essere eseguite. L'esecuzione di una regola incorpora la valutazione della condizione e l'esecuzione dell'azione.
Più precisamente un evento può essere visto come una coppia ($<$ *tipo evento* $>$,$<$ *tempo* $>$) dove $<$ *tipo evento* $>$ denota la descrizione dell'evento che causa la reazione del sistema, e $<$ *tempo* $>$ rappresenta l'istante nel tempo in cui l'evento è avvenuto. Quello che l'utente deve definire è il tipo di evento che può essere determinato a seconda del DB e del suo ambiente. Per esempio possono essere specificati come eventi l'inizio o la fine di una operazione di modifica dei dati, allora nel caso di un database relazionale, si avranno le operazioni insert, delete, update su una particolare relazione, invece, nel caso di un DB orientato agli oggetti, si avranno le operazioni di creazione, cancellazione di un oggetto, oppure l'invocazione di un metodo. Un evento può essere primitivo o composto. Evento primitivo corrisponde ad occorrenze elementari ed è strettamente connesso ad un istante di tempo determinato

dall'accadere di un evento nel DB o nel suo ambiente. Evento composto è definito come combinazione di eventi primitivi usando costruttori come disgiunzione, congiunzione, ecc. È collegato ad un istante di tempo che tiene conto delle informazioni sui componenti dell'evento composto, tipicamente l'istante dell'ultimo evento occorso. Alla coppia di paramentri ($< tipo\ evento >$,$< tempo >$) se ne possono aggiungere altri a seconda delle necessità, come per esempio: quale transazione ha causato l'evento, quale utente ha iniziato la transazione, ecc.
L'event history o *history* consiste nella storia di tutte le occorrenze degli eventi definiti, esiste in qualunque database attivo. La storia comincia nel momento in cui il primo evento è definito e può essere mantenuta per più sessioni poiché per gli eventi composti si può aver bisogno della storia delle sessioni precedenti.
La condizione è una formula che deve essere verificata perché l'azione sia eseguita. La valutazione della regola deve essere fatta quando la regola è triggered. In particolare la formula è un predicato che interroga il database (come ad esempio la clausola where dell'SQL) e deve essere vera o deve ritornare un insieme di oggetti non nullo.
L'azione descrive la reazione all'evento ed è eseguita quando la regola è attivata e la condizione è vera. Può contenere modifiche dei dati, operazioni di recupero dei dati, operazioni su transazioni come commit o abort, chiamate a procedure o metodi, ecc. Il fatto che l'azione può fare operazioni sui dati determina la possibilità di attivare altre regole causando il così detto *triggering a cascata* che se sottovalutato può portare il sistema in uno stato ciclico senza uscita.
In analogia al DDL un ADBMS fornisce un Rule Definition Language (RDL) che serve per specificare in modo completo le regole ECA. Il linguaggio consiste in costruttori per la definizione delle regole, eventi, condizioni, azioni e per l'esecuzione. In aggiunta al RDL e ad un sistema per rilevare gli eventi un ADBMS deve supportare l'esecuzione delle regole. E' necessario, dunque, un modello di esecuzione che mi determini quando la regola deve essere eseguita e in quale transazione, che risolva eventuali concorrenze e che gestisca il recovery.

Capitolo 2

Stato dell'arte sulle basi di dati attive

2.1 Funzionalità di un ADBMS

Le basi di dati con comportamento attivo sono diventate un argomento importante nelle ricerche. Molti sistemi e modelli proposti si dicono "attivi", però è ancora poco chiaro cosa il termine DBMS attivo realmente significhi. Sotto quali condizioni si può dire che il sistema è attivo?
Per rispondere a questa domanda si devono analizzare tutti i sistemi proposti, estrarre le funzionalitá comuni e stabilire quali siano necessarie e quali no.
Si analizzano ora le funzionalità che un ADBMS deve avere senza fare riferimento all'ambiente di applicazione [**?**]. Prima di tutto si descrivono le funzionalità che il sistema deve avere per essere considerato attivo.

1. Un ADBMS è un DBMS: tutti i concetti necessari per un sistema passivo sono altrettanto necessari per un sistema attivo. Questo significa che se l'utente ignora completamente le funzioni attive lavora esattamente con un DBMS.

2. Un ADBMS supporta la definizione e la gestione di regole ECA: si estende un sistema passivo supportando un comportamento reattivo. Questo comportamento deve essere definibile e specificabile dall'utente. Il fatto di definire le regole insieme con i dati è detto *Knowledge model*.

   - Un ADBMS deve fornire mezzi per la definizione di eventi, condizioni, azioni. Si richiede che le situazioni siano descritte da una

coppia (evento - condizione). Solitamente è utile lasciare al compilatore, o all'ADBMS, stesso la generazione delle definizioni degli eventi (definiti quindi implicitamente), però deve anche essere possibile definire esplicitamente gli eventi, cioè l'utente può definire eventi a suo piacimento. In generale si richiede che la caratteristica *before* e *after* event sia definita. Nel caso di operazioni del database, per esempio, un before determina una segnalazione prima che l'operazione sia veramente eseguita, un after fa in modo che la segnalazione avvenga immediatamente dopo che l'operazione è stata eseguita. Se la parte evento è obbligatoria allora la condizione può essere omessa o specificata direttamente nella parte azione.

- Un ADBMS deve supportare la gestione delle regole e l'evoluzione del set di regole. Il set di regole definite costituiscono il *rulebase*, che deve essere gestito ovunque sia salvato: nel database o altrove. Il rulebase deve poter essere cambiato, abilitato e disabilitato nelle sue parti senza conseguenza per il sistema. La disattivazione di una regola non implica la sua cancellazione, anzi, essa rimane definita nel sistema ma non può essere mai attivata.
- Un ADBMS deve avere un modello di esecuzione.
- Rilevamento degli eventi. Idealmente un ADBMS rivela gli eventi automaticamente. Gli eventi non devono essere segnalati dall'applicazione.
- Valutazione delle condizioni. Un ADBMS deve essere in grado di valutare una condizione dopo la rilevazione di un evento, dovrebbe essere possibile trasmettere informazioni dall'evento alla condizione.
- Esecuzione di azioni. Deve essere possibile passare informazioni dalla condizione all'azione e inoltre devono essere garantite la concorrenza e il recovery.
- Definizione della semantica di esecuzione. Ogni regola deve possedere proprietà che ne definiscono il comportamento e le caratteristiche. Queste verranno trattate con maggior chiarezza più avanti. In alcuni sistemi l'utente può scegliere quali caratteristiche applicare e quali no, in altri sistemi invece la scelta è imposta dal sistema stesso.
- La risoluzione dei conflitti deve essere predefinita o definita dall'utente. Cioè, in caso di concorrenza, il sistema deve essere capace

di stabilire quale regola deve essere eseguita per prima, rispettando un ordine che è determinato o dal sistema o dall'utente. Inoltre il sistema deve segnalare se una o più regole sono inconsistenti.

Ora si possono prendere in considerazione le funzionalità opzionali che un sistema attivo può fornire.

1. Un ADBMS può rappresentare le informazioni sulle regole in termini di modello dei dati, cioè con un linguaggio DDL opportunamente esteso, l'utente può descrivere sia il modello dei dati che le regole che ne determinano il comportamento. In questo modo non è necessario imparare un nuovo linguaggio.

2. Un ADBMS può supportare un ambiente di programmazione. Per aiutare l'utente nella creazione e gestione delle regole si possono implementare alcuni strumenti, come per esempio:

   - un browser per le regole
   - uno strumento per la progettazione delle regole
   - un analizzatore per il rulebase
   - un debugger
   - uno strumento di mantenimento delle regole
   - uno strumento per tenere traccia degli eventi e delle esecuzioni
   - uno strumento per misurare le performance del rulebase

   Questi strumenti sono separati dall'ADBMS.

3. Un ADBMS può essere ottimizzato. Può risultare utile uno strumento che misuri la performance del rulebase in modo da mostrare all'utente se la struttura è migliorabile oppure no. In altre parole è utile avere uno strumento di progettazione di rulebase come quelli per i dati in un DBMS passivo.

2.2 Caratteristiche delle regole attive

Si esaminano ora più da vicino le caratteristiche che differenziano i comportamenti delle regole.

2.2.1 Granularity

Le modifiche in un database sono generalmente classificate in *tuple* (o *instance*) *oriented* e *set oriented* a seconda che l'obiettivo della modifica sia una singola tupla o un set di queste. Questa distinzione viene estesa anche alle regole che reagiscono agli eventi causati dalle modifiche al database, perciò è possibile parlare di regole orientate alle tuple o a set di tuple a seconda che reagiscano alla singola modifica oppure ad un gruppo di modifiche. Questa caratteristica non è confinata ai database relazionali ma si estende anche a quelli orientati agli oggetti, infatti la regola può reagire ad una singola modifica di un oggetto, oppure può essere attivata una volta sola con una modifica ad una classe (che implica la modifica di tutti i suoi oggetti). Bisogna comunque sottolineare che una base di dati con modifiche set oriented non implica regole set oriented.

Esempio 1 *Prendiamo in considerazione l'SQL3 che implementa la caratteristica set oriented inserendo FOR EACH STATEMENT, mentre implementa tuple oriented con FOR EACH ROW.*

```
CREATE TRIGGER Conta_bolle
 AFTER INSERT ON ordini_evasi
 FOR EACH ROW
 WHEN TRUE
 update conta_bolle
  set bolle_nuove = bolle_nuove + 1;
```

"Ogni volta che inserisco una nuova bolla in ordini_evasi, si va ad incrementare il valore di bolle_nuove"

2.2.2 Coupling Mode

Descrive come l'attivazione della regola, la valutazione della condizione e l'esecuzione dell'azione sono sincronizzate. L'evento - condizione coupling mode (EC) descrive la relazione che esiste tra l'evento scatenante prodotto dalla transazione e la valutazione della condizione. Due differenti casi si possono accettare:

- *Immediate* : la valutazione della condizione è fatta immediatamente dopo che la transazione produce l'evento scatenante.

- *Delayed* : la valutazione della condizione non viene fatta immediatamente dopo la conclusione della parte della transazione che ha causato

l'evento, ma è spostata nel tempo fino a che non avviene un altro evento come per esempio il tentativo della transazione di fare il commit. (Quest'ultimo caso in particolare è chiamato *Deferred* EC coupling).

La condizione - azione coupling mode (CA) descrive la relazione tra la valutazione della condizione e l'istante in cui viene eseguita l'azione che prevede la regola. Sono applicabili le stesse opzioni citate prima per l'evento - condizione, quando l'azione viene eseguita al termine dell'applicazione viene, anche in questo caso, chiamata deferred.

2.2.3 Atomicità nell'esecuzione delle regole

Quando un evento di triggering è generato nella parte azione di una regola, il gestore delle regole può reagire in due modi: si sospende la regola che ha causato l'evento per eseguire la nuova regola attivata, oppure l'evento di triggering è congelato in attesa che la regola finisca la sua esecuzione. Nel primo caso si dice che la regola può essere interrotta, nel secondo, invece, si dice che è eseguita atomicamente. Da notare che, nel caso che una regola sia ricorsiva, la sua esecuzione è completata solo dopo che tutte le regole ricorsive chiamate sono state processate.

Esempio 2 *Consideriamo le due regole scritte con la sintassi di Postgres.*

```
define rule Propagate_Joe's_salary
 on replace to EMP.salary where
  EMP.name = ''Sam''
 then do
  replace EMP (salary = E.salary) where
   EMP.name = ''Sam''
  replace = ''Sam'' and EMP.name = ''Bob'';

define rule Raise_Sam's_salary
 on replace to EMP.salary where
  EMP.name = ''Sam'' and EMP.salary < 5000
 then do
  replace EMP (salary = 1.1 * NEW.salary) where
  EMP.name = ''Sam'';
```

Dopo che la prima regola è stata eseguita Sam e Bob hanno lo stesso salario, però la modifica del salario di Sam fa scattare la seconda regola che modifica a sua volta il salario di Sam del 10% se il suo salario è <5000. Questo accade se le regole sono eseguite in modalità atomica. Se invece le regole possono

essere interrotte, come in Postgres, allora subito dopo la modifica del salario di Sam, la prima regola viene interrotta per eseguire la seconda. Allora alla fine dell'esecuzione delle due regole Sam e Bob avranno salario uguale.

2.2.4 Relazione tra regole e transazioni

Con "relazione tra regole e transazioni" si vuole indicare la relazione tra la transazione che scatena la regola e la transazione che esegue la regola. Si possono avere due casi differenti. Stessa transazione; la condizione è valutata e l'azione è eseguita nella stessa transazione dove è avvenuto l'evento di triggering. Questo implica che un eventuale abort della regola produce un abort della transazione pricipale. Transazioni diverse; la valutazione della condizione e/o l'esecuzione dell'azione, sono eseguite in una transazione differente da quella che ha scatenato la regola. Questo implica che l'ordine in cui la condizione è valutata e l'azione è eseguita non è sotto la responsabilità del gestore delle regole ma del controllo della concorrenza del sistema. Il secondo caso prevede altri due sottocasi. Sottotransazioni dipendenti: la transazione per l'esecuzione della regola è una sottotransazione di quella dove è avvenuto l'evento di triggering. Anche in questo caso un fallimento della regola implica un fallimento della transazione principale. Transazioni indipendenti: le due transazioni non sono collegate tra di loro e quindi il fallimento di una non implica il fallimento dell'altra. Nel proseguo si tratteranno solo i casi che contemplano la stessa transazione per la regola e l'evento scatenante.

2.2.5 Risoluzione dei conflitti

Nei sistemi possono essere attive simultaneamente più regole, o perché un evento può attivarne più di una, oppure perché le regole non sono considerate immediatamente dopo l'evento scatenante ma ritardate o, infine, perché una regola che produce eventi di triggering viene eseguita in modo atomico e quindi gli eventi da essa generati vengono considerati alla fine della sua esecuzione. Perciò si necessita di una politica di risoluzione dei conflitti per decidere quale regola eseguire per prima. Le alternative possibili sono :

- Esecuzione seriale: il sistema sceglie una regola e la esegue. La scelta viene determinata dalla priorità che può essere interamente data dall'utente oppure può essere data dal sistema stesso. Un esempio è fornito da Postgres nel quale il gestore delle regole deve inserire la priorità quando definisce la regola. Questo comporta la conoscenza del comportamento di tutte le regole.

- Esecuzione parallela: le regole vengono eseguite tutte in parallelo, la risoluzione dei conflitti viene lasciata al sistema.

Un altro problema che affligge tutti i sistemi attivi è quello dei cicli infiniti. La possibilità che una regola attivi altre regole può generare instabilità nel sistema. Per risolvere questo problema i sistemi commerciali offrono pochi strumenti. Uno di questi può essere un debugger che aiuta il progettista ad individuare eventuali loop. Nella maggior parte dei casi, però, i sistemi commerciali offrono solo delle limitazioni, come per esempio l'impossibilità che una regola attivi se stessa sia direttamente che indirettamente, oppure la presenza di un contatore che limita il numero di loop possibili. Un esame più approfondito di questa tematica sarà trattato più avanti nel capitolo 8.

2.2.6 Consumo degli eventi

Quando una regola viene eseguita l'evento scatenante può seguire due strade differenti chiamate modi di consumo dell'evento. L'evento può mantenere la capacità di attivare altre regole e, nel caso che ciò non avvenga ci si chiede quando questo evento può dirsi consumato. Determiniamo innanzi tutto lo scopo del consumo dell'evento. Si possono avere tre possibilità :

- *No consumption*: l'evento scatenante non risente della valutazione della condizione o dell'esecuzione della regola, in particolare mantiene la possibilità di attivare altre regole. Si può dire che la regola rimane attivata fino a che la condizione non diventa falsa.

- *Local consumption*: l'evento perde la possibilità di attivare nuovamente la stessa regola appena eseguita ma conserva la possibilità di attivare altre regole non ancora considerate. (Questa opzione è quella più frequentemente usata nei sistemi esistenti).

- *Global consumption*: l'evento perde la possibilità di attivare altre regole.

Il consumo dell'evento in termini di tempo, cioè quando l'evento viene cancellato dal sistema perché già preso in considerazione, può avvenire subito dopo la valutazione della condizione, oppure subito dopo l'esecuzione della regola.

Esempio 3 *L'esempio riportato utilizza la semantica dello Starburst che nella prima versione consuma l'evento a livello dell'esecuzione, mentre nella seconda versione il consumo dell'evento avviene a livello della condizione.*


```
create rule Created_Carlo
 when inserted Persona
 if exist select * from inserted where name=''Carlo''
 then print (``Una persona di nome Carlo \`e stata inserita'');
```

Nella prima versione dello Starburst questa regola può essere attivata da un inserimento di una persona nella tabella Persona e rimanere attiva per tutta la transazione anche se l'inserimento non riguarda nessuna tupla con nome uguale a Carlo. Nella seconda versione, dove il consumo dell'evento avviene al momento della considerazione, la regola viene esclusa dopo la prima valutazione negativa.

2.2.7 Informazioni sulla storia delle transazioni

In molti sistemi è possibile che le regole richiedano informazioni esplicite riguardanti la storia delle transazioni. Due tipi di informazioni possono essere richieste:

- *Past Data* cioè informazioni sui valori passati dei dati, normalmente ciò è possibile estendendo il linguaggio usato dalle regole nelle query. Solo particolari stati possono essere esaminati: lo stato prima dell'esecuzione della transazione (*pre transaction*); lo stato dopo l'ultima considerazione della regola (*last consideration*); lo stato prima dell'occorrenza dell'evento che ha scatenato la regola (*pre-event*).

- *Past Event* cioè informazioni sugli eventi passati causati dalle transazioni e sugli oggetti toccati da questi eventi. In questo modo una regola è valutata solo su quei dati che sono stati effettivamente modificati anche da transazioni precedenti.

Esempio 4 *Per il primo caso si può considerare una regola scritta in SQL3 dove lo stato passato di una intera relazione può essere esaminato.*

```
CREATE TRIGGER Conta_bolle
AFTER INSERT ON Ordini_evasi
REFERENCING OLD_TABLE AS Old_ord
FOR EACH STATEMENT
WHEN TRUE
update conta_bolle set bolle_nuove=bolle_nuove+
                (select count(*) from Ordini_evasi -
                  select count(*) from Old_ord)
```

Nel secondo caso si fa riferimento ad una regola scritta con la sintassi del DB orientato agli oggetti Chimera. La sua funzione è quella di bloccare tutti gli utenti che falliscono il login per tre volte. L'evento create(Intruder-l-r) in una transazione segnala che è avvenuto un tentativo di intrusione. Tramite la creazione degli oggetti Intruder-l-r la regola monitorizza il numero dei tentativi falliti.

```
define event_preserving, immediate trigger Chimera_report
 event create(Intruder-l-r)
 condition Intruder-l-r(x),occurred(create(Intruder-l-r, x)),
    integer(y), y=card(x), y>3 person(z), z=x.user
 actions modify(User,Status,z,''locked'')
end;
```

2.2.8 Eventi composti

Alcuni sistemi danno la possibilità di attivare regole attraverso un'arbitraria combinazione di eventi primitivi (update, comandi di transazioni, segnali del clock, ecc.). L'occorrenza di un determinato evento può essere vista anche come invalidante di eventi avvenuti precedentemente, in questo caso si considera l'effetto che l'evento ha nella sequenza (Net Effect). Per esempio si può creare un'entità, modificarla più volte e poi cancellarla, in questo caso nessuna regola dovrebbe essere attivata perché il sistema non subisce variazioni. Normalmente la reazione della regola avviene dopo l'evento scatenante e l'effetto della regola stessa dopo la modifica dell'evento. In alcune applicazioni però sarebbe importante che l'effetto della regola si vedesse prima della modifica dell'evento o al suo posto, per esempio negli *alerts*.

Esempio 5 *Nel caso di regole scritte con Chimera, è possibile ottenere un comportamento di tipo Net Effect utilizzando holds al posto di occurred nelle interrogazioni agli eventi.*

```
define event_con, immediate
 trigger Chimera_monitor
 events modify(Emp.salary)
 condition Emp(x), holds(modify(Emp.salary), x),
   x.salary >1.1*old(x.salary)
 actions modify(Emp.salary, x, 1.1*old(x.salary))
end;
```

Come già accennato in precedenza tutte queste caratteristiche possono essere scelte dall'utente solo in alcuni sistemi. Nella maggior parte dei casi è il sistema stesso che sceglie per l'utente.

Capitolo 3

Sistemi ADBMS prototipali e commerciali

3.1 I diversi approcci architetturali

Esistono diversi tipi di implementazione per gli ADBMS :

- Implementazione da zero.
- Architettura integrata.
- Architettura stratificata.

Nel primo approccio, l'ADBMS è costruito completamente da zero, anche le parti passive devono essere implementate. È inevitabilmente più costoso degli altri due approcci.
La seconda possibilità è quella di utilizzare un DBMS passivo, modificarlo ed estenderlo internamente. L'operazione di modifica necessita però del codice sorgente e di una elevata conoscenza dell'architettura del DBMS utilizzato. Per sistemi complessi come i database ciò è molto difficile da ottenere, senza una stretta collaborazione con i progettisti e gli sviluppatori del sistema passivo. Esistono però anche dei DBMS-Toolkit che definiscono un modello di architettura e un set di librerie che implementano certe funzionalità dei DBMS. Se la funzionalità che cerchiamo non è presente nella libreria del Toolkit, è possibile crearla ex-novo e con poco software inserirla nel DBMS.
L'ultima possibilità è quella di usare un DB passivo esistente e implementare il comportamento attivo sopra questo sistema. In contrasto con gli altri approcci, il sistema utilizzato è visto come una scatola nera e non può essere modificato internamente.

Il primo lavoro su architetture di ADBMS è stato fatto nel progetto HiPac [?] all'inizio degli anni 80. Da allora sono state proposte molte definizioni di linguaggi per regole e molte tecniche di rilevamento degli eventi.
Postgres [?] e Starburst [?] sono ADBMS relazionali ottenuti aggiornando la versione passiva degli stessi sistemi. Se consideriamo l'architettura stratificata si può citare A-RDL [?], come sistema relazionale, ACOOD (Active Object-Oriented Database System) [?] e TriGS (Trigger system for GemStone) [?] come sistemi orientati agli oggetti.Di questi ultimi due, il primo è costruito sopra il database Ontos DB, mentre il secondo estende le funzionalità di Gem Stone. Esempi di ADBMS integrati orientati agli oggetti, ottenuti con la modifica interna del DBMS passivo, sono NAOS(Native Active Object System) [?] e Ode [?]. Il primo estende OODBMS O_2, il secondo è un altro esempio dove i DBMS passivi e attivi sono implementati dallo stesso gruppo di sviluppatori.

3.2 Sistemi ADBMS

Vediamo ora alcuni dei sistemi citati nel paragrafo precedente mettendo in evidenza soprattutto la semantica delle regole attive, e alcune delle caratteristiche.

3.2.1 Starburst

Starburst è un prototipo di DBMS (relazionale) estendibile sviluppato presso il laboratorio di ricerche IBM di Almaden (California). La sintassi delle regole è basata su SQL, è set oriented e l'elaborazione delle regole è completamente integrata con l'eleborazione delle query e la gestione delle transazioni. Gli eventi che possono scatenare le regole sono quelli di modifica dei dati; insert, delete, update. L'effetto cumulativo della successione di stati entra in gioco nell'attivazione delle regole. Si ha quindi la gestione dei *net effects*, una modifica di T seguita dalla cancellazione di T non scatena nessuna regola associata ad update(T).
La sintassi delle regole è:

```
create rule nome on tabella
when predicato di transazione
[ if condizione ]
then lista di azioni
[ precedes lista di regole ]
[ follows lista di regole ]
```

Il *predicato di transazione* specifica gli eventi che scatenano la regola e possono essere *insert, delete, update e update(c0,...cn)*. Può anche essere accettata una disgiunzione di eventi, cioè X or Y or Z ,ecc. (eventi composti).
La *condizione* è un predicato SQL. L'*azione* è una qualsiasi operazione sulla base di dati insert, rollback, create table, ecc. Infine *precedes* e *follows* permettono di specificare le precedenze fra le regole per la risoluzione dei conflitti. Le parti tra parentesi quadre sono opzionali.
Per implementare la storia delle transazioni, vengono generate delle tabelle temporanee, che sono poi richiamate sia nella condizione, che nell'azione. Le regole vengono elaborate nei "punti di elaborazione". Uno di questi punti è la fine della transazione, oppure è l'utente stesso che può definirli.
Concludendo, Starburst può vantare una implementazione completa ed integrata, una granularità di esecuzione delle regole flessibile, un potente linguaggio di espressione delle condizioni (SQL) e la possibilità di ordinare le regole stesse. Però ha un insieme ridotto di eventi scatenanti (solo operazioni di modifica dei dati), un solo modo di esecuzione all'interno di una transazione e, inoltre, non esiste nessuna interazione con le applicazioni, cioè non è possibile passare parametri alle applicazioni.

3.2.2 Postgres

Postgres è un prototipo di DBMS (relazionale) di nuova concezione sviluppato all'Università di Berkeley (California).
La sintassi delle regole è basata su Quel ed è una caratteristica fondamentale del DBMS, infatti viene utilizzata per realizzare viste, versioni, sicurezza, ecc. Le regole sono tuple oriented, cioè le condizioni di modifica sono basate su modifiche a livello di tupla. Le regole possono essere attivate non solo da eventi di modifica come insert, delete o update, ma anche da un retrive, cioè da una ricerca di dati in lettura. Nell'azione può essere specificata anche una operazione alternativa da eseguire al posto di quella della transazione se la condizione è verificata (instead).
La sintassi è:

```
define rule nome
on evento to oggetto
[[ from lista di tabelle ] where condizione ]
then do [ instead ] lista azioni
```

Dove l'*oggetto* è una tabella o una colonna, l'*evento* specifica l'azione scatenante e può essere append, delete, replace (update), retrive. La *condizione* è un predicato Quel sulla tuple innescante e su tabelle specificate nella clausola

from. L'*azione* è una qualunque operazione sulla base dei dati. Si può far riferimento a valori new/current per la storia del database.
Tutte le regole sono elaborate per mezzo di transazioni. Non esiste un meccanismo di ordinamento, l'ordine è arbitrario.
L'implementazione delle regole può essere fatta in due modi:

1. *marking*: si mettono indicatori su tutte le tuple per cui si applica la regola, quando durante l'esecuzione si incontra un marker si attiva il processore delle regole che controlla la condizione e lancia l'azione corrispondente. I markers possono essere portati a livello di tabella.

2. *rewrite*: è appropriato quando poche regole sono associate a molte tuple. Quando si fa una query da terminale, questa viene associata con le regole ed ottimizzata per un'esecuzione più veloce (per alcune combinazioni regola-query può non funzionare).

Concludendo Postgres può essere molto utile per alcune applicazioni ed ha una forte somiglianza con SQL3, però permette l'esecuzione della regola solo al livello di tupla (tuple oriented), ha una semantica e una implementazione approsimative e, infine, non c'è nessuna interazione con le applicazioni.

3.2.3 HiPAC

(High Performance Active Database System)
HiPAC è un prototipo di DBMS attivo orientato agli oggetti sviluppato a CCA/XAIT (Xerox).
Le regole sono rappresentate come oggetti, hanno struttura e operazioni e si può applicare il concetto di ereditarietà.
Struttura e operazioni delle regole:

```
Class regola :Subclass of oggetto
Struttura
Evento :
Condizione
Azione
Altri attributi
Operazioni
Create
Delete
Modify
Enable
```

```
Disable
Fire
```

Le regole possono essere create, lette, modificate, cancellate come gli altri oggetti sotto il controllo dell'accesso concorrente e della sicurezza.
Anche gli eventi sono degli oggetti a tutti gli effetti, e possono essere di vario tipo: eventi sulla base dei dati come chiamate di metodi, read, create, abort, ecc, oppure eventi temporali come: assoluti (es: il giorno 22/01/99), relativi (es: 10 minuti dopo il metodo Elabora), periodici (es: alle 5:00 di ogni domenica). Inoltre gli eventi possono essere composti, dagli operatori or, sequence, clasure, oppure parametrizzati , e i parametri sono passati alla condizione e all'azione.
La condizione si compone di due parti:

1. un insieme di queries; la condizione è vera se tutte le query ritornano un risultato non nullo. Il risultato è memorizzato e passato all'azione.

2. un modo di accoppiamento, che specifica quando la condizione è valutata rispetto alla transazione in cui è stato generato l'evento. (EC coupled mode : immediate, deferred, decoupled)

L'azione consiste anch'essa in due parti :

1. un programma arbitrario

2. un modo di accoppiamento che specifica quando l'azione è eseguita rispetto alla transazione in cui è stata valutata la condizione.(CA coupling mode : immediate, deferred,decoupled)

Concludendo HiPAC può vantare un'integrazione con il metodo orientato agli oggetti, un linguaggio espressivo per eventi, condizioni, azioni, i metodi di accoppiamento (EC, CA) e l'interazione con le applicazioni. Però HiPAC ha un linguaggio ostico per la specifica delle regole e, fatto più importante, non è stato implementato.

3.2.4 Ode

Ode è un prototipo di DBMS orientato agli oggetti sviluppato nei laboratori Bell di AT&T.
Le regole sono definite come estensioni del linguaggio di programmazione della base di dati O++, sono associate alla definizione delle classi e quindi si applica l'ereditarietà. Si possono dividere in : Vincoli (hanno semantica di

esecuzione diversa, è un subset dei trigger) e Trigger.
Sintassi dei Vincoli:

```
Nella definizione della classe
constraint
condizione_1 :  azione_1
...
condizione_n :  azione_n
```

La *condizione* è un predicato sulle componenti degli oggetti, cioè sugli attributi della classe.
L'*azione* è un'istruzione unica che può anche essere una chiamata ad una procedura o a un metodo. Ogni coppia condizione azione può essere eseguita immediatamente (hard) o al termine della transazione (soft) (Sono esattamente i modi di applicazione).
Sintassi dei trigger:

```
trigger
nome_1 (params_1)
evento_1 and condizione_1 => azione_1
...
nome_n (params_n)
evento_n and condizione_n => azione_n
```

l'*evento* può essere una chiamata ad un metodo, create, delete, ecc. e può avere la specifica temporale before o after. Anche un comando di una transazione può essere considerato come un evento. Infine esiste anche la possibilità di comporre questi eventi con i costrutti and, or, not e sequence.
La *condizione* è un predicato sui componenti dell'oggetto della classe. L'*azione* è un'istruzione unica che può essere anche una chiamata di una procedura o di un metodo. Tutte e due, condizione e azione, possono fare riferimento ai parametri *params*.
Esiste anche un meccanismo di *timeout* che permette di definire un intervallo di tempo entro il quale si deve eseguire l'azione delle regole, se si è fuori dall'intervallo temporale si esegue l'azione del timeout.

```
nome_1(params_1)
within espressione temporale
evento_1 and condizione_1 => azione_1
azione di timeout
```

Concludendo Ode può vantare un potente linguaggio per gli eventi, le nozioni separate di vincolo e di trigger, l'integrazione con il linguaggio di programmazione (anche se O++ è proprietario) e inoltre alcune caratteristiche utili come il time out e i modi di accoppiamento. Però non è stato ancora implementato completamente.

3.3 Confronto tra i sistemi ADBMS

Dai paragrafi precedenti si nota come tutti i sistemi, sia relazionali che orientati agli oggetti, per implementare il comportamento attivo facciano ricorso al paradigma delle regole ECA (evento-condizione-azione). Si vuole far notare inoltre come la sintassi delle regole attive sia completamente diversa da sistema a sistema, questo per sottolineare il fatto che non esiste uno standard in questo settore al quale fare riferimento. É per questo che in alcuni casi la parte condizione puó essere definita con costrutti dell'SQL, mentre in altri sono utilizzabili solo i costrutti della clausola where quindi un minor potere espressivo. Nel caso della definizione degli eventi, tutti i sistemi ADBMS implementano quelli classici: insert, update, delete; mentre ancora pochi estendono questo set ad altre tipologie di evento.
Tenendo in considerazione un panorama piú ampio di sistemi ADBMS si riporta la tabella 3.1 che mostra quali caratteristiche vengono implementate e quali no. Nella maggior parte dei casi,é facile notare come l'ADBMS stesso ad imporre un certo comportamento al rulebase. Infatti in pochissimi casi l'utente ha la possibilitá di scegliere.

Proprietà	Valore	Sistemi che la implementano
Granularity	Instance Oriented	Postgres,TriGS, Samos, Ode,Oracle, SQL3
	Set Oriented	A-RDL, Chimera, NAOS, SQL3, Starburst, Oracle
E-C Time Coupling	Immediate	A-RDL, Chimera, HiPac, Postgres, Naos Oracle, Samos, SQL3, TriGS
	Delayed	A-RDL, Chimera, HiPac, NAOS, Samos, SQL3, Starburst, TriGS
C-A Time Coupling	Immediate	A-RDL, Chimera, HiPac, Postgres, Naos, Oracle, Samos, Starburst, SQL3, TriGS
	Delayed	HiPac, Ode, Samos, TriGS
Action Execution	Atomic	A-RDL, Chimera, HiPac, Starburst
	Interruptable	NAOS, Ode, Postgres, Samos, SQL3
Consumption Scope	None	
	Local	A-RDL, Chimera, HiPac, NAOS, Ode, Oracle, Postgres, Samos, SQL3, Starburst, TriGS
	Global	Postgres
Consumption Time	At Condition	Chimera, Postgres, Oracle, Samos, Starburst II, SQL3
	At Execution	NAOS, Starburst I
Composite Event	Disjunctions	A-RDL, Chimera, Oracle
	Arbitrary	HiPac, Ode, Samos
Event Net Effect		A-RDL, NAOS, Ode, Starburst, Chimera
Conflict Resolution	Serial	A-RDL, Chimera, NAOS, Ode, Oracle, Samos, SQL3, Starburst, TriGS
	Parallel	HiPac, Ode

Tabella 3.1: Tabella riassuntiva delle caratteristiche dei sistemi ADBMS

Capitolo 4

Progetti di ricerca su ADBMS

Lo scopo di queste pagine è quello di analizzare diversi lavori per la costruzione di un ADBMS, per individuare l'approccio più adatto per estendere ODB-Tools.

4.1 Classi di regole ed eventi

All'università di Zurigo, Svizzera, è stato sviluppato un ADBMS chiamato SAMOS [?] con architettura stratificata su un DBMS passivo, che viene quindi visto come una scatola nera e non modificato. In SAMOS le regole e gli eventi sono rappresentati come oggetti, hanno un'identità e possono essere manipolati, come gli altri oggetti, tramite l'utilizzo dei metodi. Per definire le regole, SAMOS fornisce un linguaggio RDL in aggiunta al DDL per i dati. La sintassi di questo linguaggio è la seguente:

```
DEFINE RULE nome
ON evento
IF condizione
DO azione
COUPLING MODE coupling_mode
PRIORITIES (BEFORE|AFTER) nome_rule

e per definire gli eventi

DEFINE EVENT nome_evento definizione_evento
```

Le descrizioni degli eventi possono essere definite esplicitamente o implici-

tamente nelle regole e possono fare riferimento ai seguenti tipi primitivi:

- *Method Event* segnala un evento prima o dopo l'esecuzione di un metodo. L'utente accede e manipola gli oggetti spedendo dei messaggi, ogni messaggio prevede l'esecuzione di un metodo. Alla fine dell'esecuzione è il metodo che spedisce un messaggio. Quindi intercettando questi messaggi è possibile attivare delle regole. La sintassi per questi eventi è:

  (BEFORE|AFTER)"."(class_name| object_name | "*")"."method_name

  Un *method event* può essere dunque collegato:

  - ad una classe, allora l'evento è segnalato ogni qualvolta viene eseguito il metodo scelto per un qualunque oggetto appartenente alla classe.
  - ad un particolare oggetto (assumendo che sia dato un nome unico agli oggetti), allora l'evento è segnalato ogni volta che viene eseguito il metodo per quell'oggetto.
  - a tutte le classi (nel caso di "*"), allora l'evento viene segnalato ad ogni esecuzione del metodo per un qualunque oggetto di una qualunque classe che contiene quel metodo.

  Se per esempio si vuole segnalare un evento prima dell'esecuzione del metodo che calcola l'età di una persona, si dovrà definire l'evento come *DEFINE EVENT E1 BEFORE.Persona.calc_età*. Se invece il metodo che scatena la regola appartiene a più classi allora deve definirlo in quest'altro modo *DEFINE EVENT E2 AFTER.*.stampa*.

- *transaction event* segnala un evento prima o dopo un'operazione di una transazione (begin, commit, abort). Assumendo che le transazioni abbiano un nome, la sintassi è:

  (BOT | EOT | ABORT) [transaction_name]

- *time event* segnala un evento nel momento specificato, oppure periodicamente o, infine, dopo un certo intervallo di tempo dall'occorrenza di un evento. La sintassi è:

  EVERY frequency (YEAR|MONTH|WEEKDAY|HOUR|MINUTE) time [WITHIN interval]

Un esempio può essere *EVERY 7 DAY 22:00*, l'evento viene segnalato ogni 7 giorni alle 22:00. *Interval* è opzionale e rappresenta l'intervallo entro il quale deve verificarsi l'evento, per esempio *EVERY 7 DAY 22:00 WITHIN [05.01-06.22]*, l'evento è segnalato ogni 7 giorni solo tra il primo di Maggio e il 22 di Giugno.
Nel caso si voglia definire un *time event* in relazione ad un altro evento allora si ottiene un evento al tempo *t+x*, dove t è l'istante in cui è stato segnalato l'evento, x è il ritardo dato dal *time event*. Questo meccanismo si capirà meglio più avanti quando si parlerà dei parametri degli eventi.

- *abstract event* segnala un evento quando richiesto esplicitamente da un utente esterno o da una applicazione.

  `DEFINE EVENT event_name`

  SAMOS non rivela questi eventi ma devono essere segnalati tramite

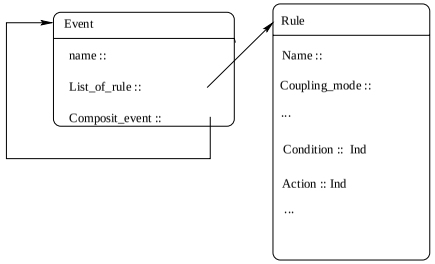
  `RAISE EVENT event_name`
  Nel momento in cui uno di questi eventi viene definito e poi segnalato viene trattato come tutti gli altri eventi sopra specificati.

La descrizione dell'evento comprende anche dei parametri che forniscono informazioni sull'istante dell'occorrenza, sulla transazione che ha dato luogo all'evento, sull'oggetto al quale è stato mandato il messaggio, se si tratta di un evento metodo, e infine anche sull'user che ha eseguito la transazione. Se voglio creare un *time event* riferito ad un evento basta sommare la quantità x (sopra specificata) al parametro che memorizza l'istante di occorrenza dell'evento.
Dopo che un evento è stato rilevato, le condizioni delle regole associate a quell'evento vengono verificate. Per ogni condizione vera viene eseguita l'azione. Entrambe, condizione e azione, devono essere specificate nel DML del DBMS.
Infine, SAMOS fornisce anche la possibilità di definire eventi composti mettendo a disposizione dell'utente alcuni costruttori. La disgiunzione (E1|E2) che segnala l'evento all'occorrenza di E1 o E2. La congiunzione (E1,E2) che segnala l'evento all'occorrenza di E1 e di E2 (non è importante che rispettino l'ordine). In una sequenza (E1;E2) i due eventi devono avvenire, questa volta, in ordine di occorrenza. Quando invece si definisce un evento composto con questa sintassi (*E IN I) viene segnalato solo la prima occorrenza di E durante tutto l'intervallo I. (TIMES(n,E) IN I) segnala un evento ogni volta che l'evento E occorre n volte nell'intervallo I. Infine c'è la possibilità di

definire eventi negativi, infatti (NOT E IN I) mi segnala un evento se E non occorre mai in tutto l'intervallo.

Descrizione degli eventi

Tutte le descrizioni degli eventi definiti dagli utenti sono istanze della classe evento. Gli oggetti evento possono essere creati, modificati, acceduti attraverso i metodi implementati nella classe *Event*.



Per ogni tipo di evento è possibile definire una sottoclasse che contenga tutti gli attributi necessari. Ogni evento ha l'attributo *nome* che lo differenzia dagli altri, ha l'attributo *list of rules* che specifica quali regole devono essere eseguite ad ogni occorrenza dell'evento (è un collegamento alla classe delle regole) e, infine, ha l'attributo *composit event* che mi permette di implementare gli eventi composti (è un collegamento alla classe degli eventi). In SAMOS quindi si associano le regole con la descrizione degli eventi, questo approccio è più generale perché può essere usato sia con gli eventi primitivi che con quelli composti. Se, invece, si associano le regole alle classi e agli oggetti per cui sono definite, è possibile utilizzare solo eventi primitivi, perché solo dopo la loro occorrenza può avvenire l'associazione con la regola.

Descrizione delle regole

Le definizioni delle regole sono istanze della classe *Rule*. Gli attributi di questi oggetti comprendono un nome, tutte le caratteristiche spiegate in precedenza (Coupling mode, granularity, ecc.), la condizione, e la parte azione. In SAMOS questi due ultimi attributi sono in sostanza due puntatori che identificano le procedure che eseguono le richieste della condizione ed eventualmente eseguono i comandi dell'azione. Non è possibile infatti inserire come attributi delle query o dei comandi. Tutte le procedure di condizione

e di azione sono raccolte in due file che devono essere controllati e compilati dal sistema.
Si sottolinea che con questo approccio non è indispensabile avere a disposizione un linguaggio RDL per definire le regole ECA. Infatti l'utente può definire le regole direttamente implementando gli oggetti regola ed evento eseguendo i metodi offerti dal sistema. Però la possibilità di utilizzare un linguaggio di definizione rende sicuramente la gestione del sistema più semplice e più sicura. Infatti la traduzione della definizione agli oggetti viene fatta in modo automatico da un compilatore che rivela immediatamente gli eventuali errori di definizione. Inoltre il compilatore può effettuare la traduzione della condizione e dell'azione direttamente in metodi.

4.2 Regole ECAA e Rule Service in CORBA

Le regole nei sistemi degli ADBMS sono generalmente specificate nella forma ECA o ECAA [?]. La sintassi generale del linguaggio delle regole è la seguente.

```
EVENT couplingMode(TriggeringEvent)
CONDITION cond Expression
ACTION operation(s)
ALTERNATIVE ACTION operation(s)
```

La clausola *Event* fornisce le specifiche dell'evento di trigger. Include il coupling mode e trigger event. Il primo specifica il tempo in cui la regola deve essere attivata, alcuni esempi sono: Before, After, AfterTransaction, On, When, At, Detached, etc. Il *triggering event* può essere ogni evento legittimato che è definito per il modello/linguaggio dei dati con il quale il linguaggio delle regole è associato. I tipi di evento più importanti che possono essere definiti sono:

- Invocazione di un metodo
- Esplicito invio di un evento

Esempio 6 *È possibile definire una regola del tipo pre-condizione, "Prima dell'esecuzione del metodo delete, se l'oggetto è usato, non avviare il metodo"*

```
rule no_cancel is
EVENT before Part::delete()
CONDITION this.beingUsed()
ACTION abort;
end;
```


Associate alla specifica dell'interfaccia di ogni classe, ci possono dunque essere delle regole ECAA che agiscono sul comportamento degli oggetti della classe. Si possono ora mettere in evidenza alcuni punti importanti:

- L'implementazione di un metodo deve eseguire solo le funzioni alle quali è dedicata (es la funzione delete() deve solamente cancellare l'oggetto e non verificare se l'oggetto è ancora utilizzato).
- La semantica della pre-condizione e della post-condizione non sono comprese nella implementazione del metodo. Invece devono essere esplicitamente specificate nelle regole ECAA.
- Se la politica di gestione a cui si riferiscono la pre e la post condizione cambia solo, l'ECAA deve cambiare, l'implementazione del metodo non deve essere modificata.

Esempio 7 *Questo esempio mostra l'uso di una regola ECAA per eseguire un'azione in risposta a un evento generato da una transazione o da una applicazione. L'evento EC-event è definito dall'utente.*

```
rule EC-event is
EVENT onEvent EC-event
CONDITION (ec.part.projactType='g')
ACTION ec.addToECPackage();
end;
```

Questo esempio mostra come una regola possa essere collegata ad un evento definito dall'utente. Dopo aver definito le regole, queste devono essere eseguite dal sistema. Per questo sono necessari un rilevatore di eventi,che monitorizza il sistema durante il suo funzionamento, e un processo che verifichi la condizione ed esegua eventualmente l'azione. Quest'ultimo componente è chiamato *rule services*. Le sue funzioni principali sono quelle di definire, cancellare, rimpiazzare, attivare, disattivare e ritrovare le regole.

In questo caso quindi non si definiscono nuove classi nel db ma si gestiscono le regole con componenti esterni, quali il monitor degli eventi e il rule services.

La semantica di definizione delle regole è un linguaggio RDL da aggiungere al DDL e al DML del DB. Non è un'estensione del DDL.

4.3 Regole EECA

Un altro studio sulla semantica dei ADBMS che può essere considerato è quello sviluppato da Fraternali e Tanca [**?**]. Il progetto ha come scopo quello di creare una semantica generale che raccolga tutte le proprietà delle regole attive, in modo tale che sia possibile tradurre le regole dei vari sistemi in un unico linguaggio, confrontando così i diversi comportamenti. Le regole scritte in EECA (Exetended Event Condition Action) sono composte dalle tre parti tradizionali chiamate: evento, condizione, azione; in più si aggiunge una parte dove vengono specificate tutte le caratteristiche menzionate in precedenza. Quindi le regole cominciano con una parte dichiarativa formata dalle parole chiave che ora vengono descritte.

- *Granularity*: che indica il comportamento della regola; Instance-Oriented (I/O) o Set-oriented (S/O).
- *Coupling Mode*: (EC) può essere *immediate* o *deferred*.
- *Atomicity*: indica se la regola è eseguita in modo atomico (*Interruptable False*) oppure è possibile che venga interrotta (*Interruptable True*).
- *Event consumption*: due parole chiave con varie combinazioni. *Consumption scope* specifica lo scopo e può assumere i valori *none, local, global*{*list of rules*}. *Consumption time* specifica l'istante in cui l'evento viene consumato e può assumere i valori *consideration* ed *execution*.

Alla parte dichiarativa segue la parte evento dove si devono specificare gli eventi che attivano la regola. Se si considerano i net effect allora la parola chiave net deve precedere la lista degli eventi.
La parte dedicata alla condizione è una query al DB oppure alla storia delle transazioni. In teoria si può adottare un qualunque linguaggio per esprimere le query, l'importante è che le regole siano corrette, sicure e che le variabili usate nella parte azione siano limitate a quelle usate nella parte condizione. In ultimo si deve considerare la parte azione. È un insieme di blocchi separati da un ";". I blocchi sono eseguiti atomicamente quindi la regola può essere interrotta solo tra un blocco e l'altro. Al loro interno i blocchi sono composti di query o operazioni di modifica separate da un punto e virgola. Il

linguaggio utilizzato non è importante come nella parte condizione. Qualunque linguaggio utilizzato deve comunque essere esteso dalle seguenti parole chiave.

- *pending(E,X)* dove E è un sottoinsieme degli eventi considerati nella parte evento. Questo comando non fa altro che legare la variabile X a un identificatore di un item (oggetto o tupla) colpito da uno degli eventi della lista E. In pratica vengono considerati solo gli oggetti colpiti da eventi che non sono ancora stati processati e che sono presenti nella lista E.
- *history(E,X)* collega la variabile X ad un identficatore di un oggetto che è stato colpito da un evento contenuto in E dall'inizio della transazione (quindi anche se già stato consumato). In questo caso nella lista E, può essere inserito anche un evento che non è presente nella parte condizione.
- Per interrogazioni su valori di dati passati : *pre transaction, last--transaction, pre-event*. Questi predicati hanno tre parametri: il primo indica il parametro che si sta analizzando, il secondo è in ingresso, e mi indica a quale valore deve tendere il parametro prima specificato, il terzo si riferisce al valore passato, prima dell'evento scatenante.

Esempio 8 *Riprendiamo un esempio trattato in precedenza che blocca un user dopo tre tentativi di accesso falliti.*

```
define event_preserving, immediate trigger Chimera_report
event create(Intruder-l-r)
condition Intruder-l-r(x), occurred(create(Itruder-l-r, x)),
integer(y), y=card(x), y>3 person(z), z=x.user
actions modify(User,Status,z,''locked'')
end;
```

Tradotta in EECA risulta:

```
Define granularity S/O EC immediate interruptable False
Consumption-scope local consumption-time consideration
Rule Chimera_report_intruder
Event:  create(Intruder-l-r)
Condition:  history(create(Intruder-l-r),X),
Y=card(X), Y>3, Z=x.user
```

Action modify(User.status,Z,''locked'')

Con questo approccio non si cerca di estendere il linguaggio dei dati DDL, ma piuttosto di costruire ex novo un linguaggio RDL che sia capace di rappresentare tutte le caratteristiche che una regola di tipo ECA può avere. Con uno strumento di questo tipo è possibile verificare la validità delle regole di qualunque sistema. Con questa base è poi possibile costruire uno strumento che traduca la regola EECA nei vari sistemi automaticamente.

Capitolo 5

L'ambiente ODB-Tools preesistente

5.1 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 e' 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 e i suoi principali componenti. Questo al fine di chiarire la sua struttura e permettere al lettore una maggiore comprensione dei miglioramenti apportati con il lavoro di questa tesi.

In figura 5.1 sono rappresentati i vari moduli che compongono tale prototipo. ODB-Tools è composto da 3 moduli:

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

- OCDL-designer:
  Questo componente software consente di controllare la consistenza di

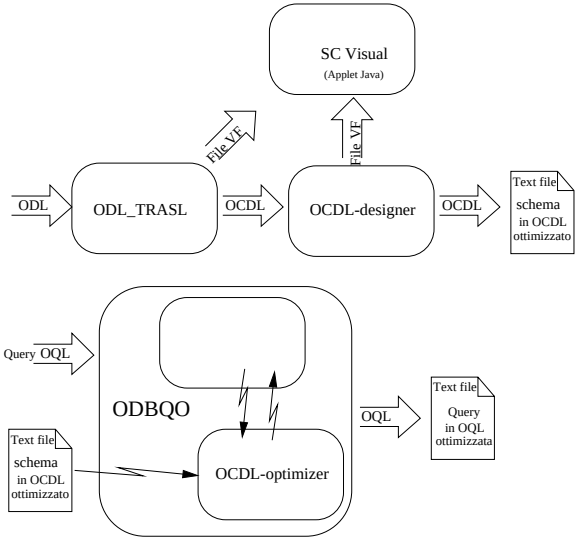


Figura 5.1: Componenti ODB-Tools

  uno schema di base di dati ad oggetti e di ottenere la minimalità dello schema rispetto alla relazione **isa**.

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

5.2 Il formalismo OCDL

In questa sezione viene brevemente riportato il formalismo OCDL introdotto in [**?**] ed esteso in [**?**] con i vincoli di integrità e vengono formalmente definite la sussunzione e l'espansione semantica.

5.2.1 Schema e Istanza del Database

Sia **D** l'insieme infinito numerabile dei valori atomici (che saranno indicati con $d_1, d_2, \ldots$), e.g., l'unione dell'insieme degli interi, delle stringhe e dei booleani.

Sia $\mathbf{B}$ l'insieme di designatori di tipi atomici, con $\mathbf{B} = \{$`integer`, `string`, `boolean`, `real`, i_1–j_1, i_2–j_2, ... , d_1, d_2, ...$\}$, dove i d_k indicano tutti gli elementi di `integer`$\cup$`string`$\cup$`boolean` e dove gli i_k–j_k indicano tutti i possibili intervalli di interi (i_k può essere $-\infty$ per denotare il minimo elemento di `integer` e j_k può essere $+\infty$ per denotare il massimo elemento di `integer`).

Sia $\mathbf{A}$ un insieme numerabile di *attributi* (denotati da $a_1, a_2, \ldots$) e $\mathcal{O}$ un insieme numerabile di *identificatori di oggetti* (denotati da $o, o', \ldots$) disgiunti da $\mathbf{D}$. Si definisce l'insieme $\mathcal{V}(\mathcal{O})$ dei *valori su* $\mathcal{O}$ (denotati da v, v') come segue (assumendo $p \geq 0$ e $a_i \neq a_j$ per $i \neq j$):

$$v \ \rightarrow d \mid o \mid \{v_1, \ldots, v_p\} \mid [a_1 : v_1, \ldots, a_p : v_p]$$

Gli identificatori di oggetti sono associati a valori tramite una *funzione totale* δ da $\mathcal{O}$ a $\mathcal{V}(\mathcal{O})$; in genere si dice che il valore $\delta(o)$ è lo *stato* dell'oggetto identificato dall'oid o;

Sia $\mathbf{N}$ l'insieme numerabile di *nomi di tipi* (denotati da $N, N', \ldots$) tali che $\mathbf{A}$, $\mathbf{B}$, e $\mathbf{N}$ siano a due a due disgiunti. $\mathbf{N}$ è partizionato in tre insiemi $\mathbf{C}$, $\mathbf{V}$ e $\mathbf{T}$, dove $\mathbf{C}$ consiste di nomi per *tipi-classe base* ($C, C' \ldots$), $\mathbf{V}$ consiste di nomi per *tipi-classe virtuali* ($V, V' \ldots$), e $\mathbf{T}$ consiste di nomi per *tipi-valori* ($t, t', \ldots$).

Un *path* p è una sequenza di elementi $p = e_1.e_2.\ldots.e_n$, con $e_i \in \mathbf{A} \cup \{\triangle, \forall, \exists\}$. Con ϵ si indica il path vuoto.

$\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$[1] indica l'insieme di tutte le *descrizioni di tipo finite* (S, $S', \ldots$), dette brevemente *tipi*, su di un dato $\mathbf{A}, \mathbf{B}, \mathbf{N}$, ottenuto in accordo con la seguente regola sintattica:

$$S \ \rightarrow \ \top \mid B \mid N \mid [a_1 : S_1, \ldots, a_k : S_k] \mid \forall\{S\} \mid \exists\{S\} \mid \ \triangle S \mid S \sqcap S' \mid (p : S)$$

$\top$ denota il *tipo universale* e rappresenta tutti i valori; [ ] denota il costruttore di tupla. $\forall\{S\}$ corrisponde al comune costruttore di insieme e rappresenta un insieme i cui elementi sono *tutti* dello stesso tipo S. Invece, il costruttore $\exists\{S\}$ denota un insieme in cui *almeno* un elemento è di tipo S. Il costrutto $\sqcap$ indica la *congiunzione*, mentre $\triangle$ è il costruttore di oggetto. Il tipo $(p\colon S)$ è detto *tipo path* e rappresenta una notazione abbreviata per i tipi ottenuti

[1] In seguito, scriveremo $\mathbf{S}$ in luogo di $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ quando i componenti sono ovvi dal contesto.

con gli altri costruttori.

Dato un dato sistema di tipi $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$, uno *schema* σ su $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ è una funzione totale da $\mathbf{N}$ a $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$, che associa ai nomi di tipi la loro descrizione. Diremo che un nome di tipo N *eredita* da un altro nome di tipo N', denotato con $N \prec\!\!\mid_\sigma N'$, se $\sigma(N) = N' \sqcap S$. Si richiede che la relazione di ereditarietà sia priva di cicli, i.e., la chiusura transitiva di $\prec\!\!\mid_\sigma$, denotata $\prec$, sia un ordine parziale stretto.

Dato un dato sistema di tipi $\mathbf{S}$, una *regole di integrità* R è espressa nella forma $R = S^a \rightarrow S^c$, dove S^a e S^c rappresentano rispettivamente l'antecedente e il conseguente della regola R, con $S^a, S^c \in \mathbf{S}$. Una regola R esprime il seguente vincolo: per tutti gli oggetti v, se v è di tipo S^a allora v deve essere di tipo S^c. Con $\mathbf{R}$ si denota un insieme finito di regole.

Uno *schema con regole* è una coppia $(\sigma, \mathbf{R})$, dove σ è uno schema e $\mathbf{R}$ un insieme di regole.

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

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

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

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

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

[2] Assumendo $\mathcal{I}_{\mathbf{B}}$ funzione di *interpretazione standard* da $\mathbf{B}$ a $2^{\mathbf{B}}$ tale che per ogni $d \in \mathbf{D} : \mathcal{I}_{\mathbf{B}}[d] = \{d\}$.

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

Definizione 1 (Istanza Legale) *Una funzione di interpretazione $\mathcal{I}$ è una* istanza legale *di uno schema con regole $(\sigma, \mathbf{R})$ sse l'insieme $\mathcal{O}$ è* finito *e per ogni $C \in \mathbf{C}, V \in \mathbf{V}, t \in \mathbf{T}, R \in \mathbf{R}$: $\mathcal{I}[C] \subseteq \mathcal{I}[\sigma(C)]$, $\mathcal{I}[t] = \mathcal{I}[\sigma(t)]$, $\mathcal{I}[V] = \mathcal{I}[\sigma(V)]$, $\mathcal{I}[S^a] \subseteq \mathcal{I}[S^c]$.*

Si noti come, in una istanza legale $\mathcal{I}$, l'interpretazione di un nome di classe base *è contenuta* nell'interpretazione della sua descrizione, mentre per un nome di classe virtuale, come per un nome di tipo-valore, l'interpretazione *coincide* con l'interpretazione della sua descrizione. In altri termini, mentre l'interpretazione di una classe base è fornita dall'utente, l'interpretazione di una classe virtuale è calcolata sulla base della sua descrizione.

In questa sede si suppone che lo schema con regole sia privo di cicli. Formalmente, diremo che un nome di tipo N *dipende* dal nome di tipo N' se N' appare in $\sigma(N)$ **oppure** se esiste in $\mathbf{R}$ una regola $N \to S$ e N' appare in S. Uno schema con regole e' privo di cicli se la relazione *dipende* non contiene cicli.

5.2.2 Sussunzione ed Espansione Semantica di un tipo

Introduciamo la relazione di sussunzione in uno schema con regole.

Definizione 2 (Sussunzione) *Dato uno schema con regole $(\sigma, \mathbf{R})$, la* relazione di sussunzione *rispetto a $(\sigma, \mathbf{R})$, scritta $S \sqsubseteq_{\mathrm{R}} S'$ per ogni coppia di tipi $S, S' \in \mathbf{S}$, è data da: $S \sqsubseteq_{\mathrm{R}} S'$ sse $\mathcal{I}[S] \subseteq \mathcal{I}[S']$ per tutte le istanze legali $\mathcal{I}$ di $(\sigma, \mathbf{R})$.*

Segue immediatamente che $\sqsubseteq_{\mathrm{R}}$ è un preordine (i.e., transitivo e riflessivo ma antisimmetrico) che induce una *relazione di equivalenza* $\simeq_{\sigma\ \mathrm{R}}$ sui tipi: $S \simeq_{\sigma\ \mathrm{R}} S'$ sse $S \sqsubseteq_{\mathrm{R}} S'$ e $S' \sqsubseteq_{\mathrm{R}} S$. Diciamo, inoltre, che un tipo S è *inconsistente* sse $S \simeq_{\sigma\ \mathrm{R}} \bot$, cioè per ciascun dominio l'interpretazione del tipo è sempre vuota.

È importante notare che la relazione di sussunzione rispetto al solo schema σ, cioè considerando $\mathbf{R} = \emptyset$, denotata con $\sqsubseteq_\sigma$, è simile alle relazioni di *subtyping* o *refinement* tra tipi definite nei *CODMs* [?, ?]. Questa relazione può essere calcolata attraverso una comparazione sintattica sui tipi; per il nostro modello tale l'algoritmo è stato presentato in [?].

Definizione 3 (Espansione Semantica) *Dato uno schema con regole $(\sigma, \mathbf{R})$ e un tipo $S \in \mathbf{S}$, l'*espansione semantica *di S rispetto a $\mathbf{R}$, EXP(S), è un tipo di $\mathbf{S}$ tale che:*

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

In altri termini, EXP(S) è il tipo più specializzato (rispetto alla relazione $\sqsubseteq_\sigma$) tra tutti i tipi $\simeq_{\sigma\ \mathrm{R}}$–equivalenti al tipo S. L'espressione $EXP(S)$ permette di esprimere la relazione esistente tra $\sqsubseteq_{\mathrm{R}}$ e $\sqsubseteq_\sigma$: per ogni $S, S' \in \mathbf{S}$ si ha $S \sqsubseteq_{\mathrm{R}} S'$ se e solo se $EXP(S) \sqsubseteq_\sigma S'$. Questo significa che, dopo aver determinato l'espansione semantica, anche la relazione di sussunzione nello schema con regole può essere calcolata tramite l'algoritmo presentato in [?].

È facile verificare che, per ogni $S \in \mathbf{S}$ e per ogni $R \in \mathbf{R}$, se $S \sqsubseteq_\sigma (p : S^a)$ allora $S \sqcap (p{:}\ S^c) \simeq_{\sigma\ \mathrm{R}} S$. Questa trasformazione di S in $S \sqcap (p{:}\ S^c)$ è la base del calcolo della $EXP(S)$: essa viene effettuata iterativamente, tenendo conto che l'applicazione di una regola può portare all'applicazione di altre regole. Per individuare tutte le possibili trasformazioni di un tipo implicate da uno schema con regole $(\sigma, \mathbf{R})$, si definisce la funzione totale $, : \mathbf{S} \longrightarrow \mathbf{S}$, come segue:

$$, (S) \;=\; \begin{cases} S \sqcap (p : S^c) & \text{se esistono } R \text{ e } p \text{ tali che } S \sqsubseteq_\sigma (p{:}\ S^a) \text{ e } S \not\sqsubseteq_\sigma (p{:}\ S^c) \\ S & \text{altrimenti} \end{cases}$$

e poniamo $, \tilde{} = , ^i$, dove i è il più piccolo intero tale che $, ^i = , ^{i+1}$. L'esistenza di i è garantita dal fatto che il numero di regole è finito e una regola non può essere applicata più di una volta con lo stesso cammino ($S \not\sqsubseteq_\sigma (p{:}\ S^c)$). Si può dimostrare che, per ogni $S \in \mathbf{S}$, EXP(S) è effettivamente calcolabile tramite $, \tilde{} (S)$.

5.3 Pregi e limiti espressivi di ODB-Tools

La struttura del database e' stata espressa in ODL esteso (vedi appendice C).
In questo paragrafo si intendono evidenziare i vari aspetti di ODB-Tools specificando, per i problemi incontrati, le soluzioni adottate.

Pregi

- La parte piu' interessante riguarda i vincoli di integrita' esprimibili come regole "if ... then ...".
  Questo ha permesso di esprimere vincoli di diversa natura, ad esempio:

  - vincoli di dominio
  - vincoli di integrita' referenziale
  - vincoli che legano attributi della stessa classe, come ad esempio "if then" che si usano in ODB-Tools
  - vincoli che legano attributi complessi (come set, list ecc..)
  - vincoli che legano attributi di classi diverse: in questo caso si deve operare utilizzando la navigazione mediante la "dot notation"

  Tutti questi vincoli sono riconducibili a una forma interpretabile da ODB-Tools.

- ODB-Tools mette a disposizione un insieme molto ampio di tipi. Questo permette di esprimere piu' facilmente i tipi degli attributi in fase di progettazione, ma questo vantaggio si perde totalmente nella fase di implementazione. Ad esempio, si sono dovuti tradurre tutti i tipi range in tipi integer, in quanto UNISQL non dispone di un tipo range predefinito.

- ODB-Tools evidenzia gli errori che si verificano nel susseguirsi di modifiche allo schema di un database, durante le prime fasi di progettazione segnalando eventuali classi o regole incoerenti.

- ODB-Tools evidenzia regole superflue, permettendo al progettista di rendere piu' semplice lo schema del database.

Limiti espressivi di OCDL

- Dall'analisi del dominio applicativo sono risultati dei vincoli di integrita' piu' complessi.
  Innanzitutto ODB-Tools permette di inserire in una regola condizioni del tipo :

  ⟨attributo⟩⟨operatore⟩⟨valore⟩

  ossia e' possibile mettere in relazione un campo con un valore costante (definito nella regola). Questo e' corretto, perche' non avrebbe senso a livello intensionale confrontare due attributi, e non avrebbe nemmeno senso presentare questa situazione all'algoritmo di sussunzione.

- Si riscontra l'impossibilita' di esprimere legami complessi tra gli attributi, come ad esempio la moltiplicazione, la divisione tra campi del database.

- Si riscontra l'impossibilita' di esprimere il comportamento nella descrizione di una classe, attraverso la dichiarazione di metodi. Questo problema e' stato risolto estendendo ulteriormente la sintassi di ODL e permettendo l'inserimento di metodi, sia nella interfaccia delle classi che nei predicati delle rule.

- Si riscontra la mancanza dell'operatore logico "OR" all'interno dei predicati delle rule. Questo limite e' stato superato introducendo una rule per ogni clausola del predicato OR.

- Non e' disponibile il tipo enumerato. Per superare questa limitazione gli attributi di tipo enumerato sono stati trasformati in range di interi, in cui ogni valore intero rappresenta un elemento del tipo enumerato corrispondente.

- Un ulteriore limite e' rappresentato dalla mancanza di un costruttore che permetta di esprimere il concetto di generalizzazione. Mediante ODL-esteso e' possibile indicare che una classe eredita le proprieta' da altre classi gia' esistenti nello schema. Non vi sono strumenti per dichiarare che una classe e' la generalizzazione di altre classi gia' esistenti nello schema. Strumenti del genere possono essere molto utili nelle fasi di evoluzione di uno schema.

5.4 Estensione di OCDL con operazioni

5.4.1 Introduzione di uno schema di operazioni

Nel paragrafo 5.2 e' stata presentata la logica OCDL su cui e' gia' basato ODB-Tools, che esprime solo descrizioni strutturali. Viene ora presentata l'integrazione delle operazioni.

Sia $\mathbf{N}$ l'insieme numerabile di *nomi di tipi* (denotati da $N, N', \ldots$) tali che $\mathbf{A}$, $\mathbf{B}$, e $\mathbf{N}$ siano a due a due disgiunti. $\mathbf{N}$ è partizionato in tre insiemi $\mathbf{C}$, $\mathbf{V}$

e **T**, dove **C** consiste di nomi per *tipi-classe base* $(C, C' \ldots)$, **V** consiste di nomi per *tipi-classe virtuali* $(V, V' \ldots)$, e **T** consiste di nomi per *tipi-valori* $(t, t', \ldots)$.

Si introducono ora i tipi atti a rappresentare lo schema delle operazioni.

Sia **Att**={in,out,inout} l'insieme dei nomi dei tipi che indicano l' "attributo del parametro", (denotati da Att_1,Att_2,... e chiamati brevemente "tipi-parametro").
Tale insieme deve essere chiaramente disgiunto dall'insieme dei nomi dei tipi **N**.
I tipi-parametro servono per introdurre nel sistema dei tipi **S** il tipo "signature di operazione", denotato con S_s, che è un particolare tipo record:

$$[p_1{:}S_1 \sqcap \mathbf{Att}_1,\ p_2{:}S_2 \sqcap \mathbf{Att}_2, \ldots, p_k{:}S_k \sqcap \mathbf{Att}_k]$$

dove:
k è il numero dei parametri della operazione
p_1,p_2,...,p_k sono i nomi dei parametri formali della operazione
$S_i \in \mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ $\forall$**i=1,...k** sono i tipi dei parametri formali

Il tipo signature rappresenta interamente gli argomenti della signature di una operazione.

Nel seguito, il sistema dei tipi costruito partendo dall'insieme dei nomi dei tipi **N** unito con **Att** ed esteso con il tipo "signature di operazione" verrà, per semplicità, indicato ancora con $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$.

Per i tipi-parametro si deve estendere la definizione di interpretazione e di istanza; in particolare, per estendere il concetto di istanza, occorre estendere la funzione σ anche a tali nomi. Queste estensioni sono basate sul fatto che un tipo-parametro deve essere considerato un nome di tipo (sia classe che tipo-valore) primitivo e senza alcuna descrizione, quindi

- Interpretazione:

$$\mathcal{I}[\mathbf{Att}] \subseteq \mathcal{V}(\mathcal{O})$$

- Schema:

$$\sigma(\mathbf{Att}) = \top$$

- Istanza:

$$\mathcal{I}[\sigma(\mathbf{Att})] \subseteq \mathcal{I}[\mathbf{Att}]$$

Si introduce ora la definizione di operazione; informalmente, le operazioni sono rappresentate da quadruple contenenti:

1. la classe sulla quale sono definite
2. il nome
3. la signature, rappresentata da un tipo simile alle ennuple con l'aggiunta degli attributi dei parametri, anch'essi rappresentati da tipi
4. il tipo di ritorno

Definizione 4 (Operazione) *Una Operazione è una quadrupla* $(C_d$,**m**,T_s,**R**$)$, *dove:*
$C_d \in \mathbf{C} \cup \mathbf{V}$ *rappresenta la classe su cui e' definita l'operazione*
$\mathbf{m} \in \mathbf{M}$ *e' il nome della operazione*
$T_s \in \mathbf{S_s}$ *e' il tipo della signature della operazione*
$\mathbf{R} \in \mathbf{S}$ *e' il tipo del risultato della operazione*

I linguaggi di programmazione orientati agli oggetti prevedono la possibilita', nota come "overloading", di dichiarare due operazioni definite sulla medesima classe con lo stesso nome. Per permettere l'overloading la coppia nome e classe di definizione non sono sufficiente ad identificare una operazione. E' necessario aggiungere l'intera signature della operazione per poterla identificare.

Per generalizzare il discorso anche al caso in cui una operazione venga introdotta direttamente in una rule è stata fatta la seguente scelta: per ogni operazione dichiarata nella interface delle classi il sistema crea un identificatore univoco, detto *identificatore di operazione*, e vi associa la quadrupla che rappresenta l'operazione stessa. Si noti che una stessa operazione (stessa quadrupla) può essere associata a più identificatori.
Formalmente, si introduce $\mathbf{I_M}$, l'insieme numerabile degli identificatori delle operazioni (denotati da Op_1,Op_2,...) e si rappresenta questa corrispondenza tramite il concetto di *Schema di operazioni*.

Definizione 5 (Schema di operazioni) *Uno* schema di operazioni $\sigma_{\mathbf{M}}$ *su* $\mathbf{I_M}$ *è una funzione totale*
$\sigma_{\mathbf{M}} : \mathbf{I_M} \rightarrow (\mathbf{C} \cup \mathbf{V}) \; X \; \mathbf{M} \; X \; \mathbf{S_s} \; X \; (\mathbf{S})$
che associa ad ogni identificatore la descrizione della propria operazione.

Per maggiori informazioni sulla traduzione delle operazioni in OCDL si veda la tesi di S. Riccio [**?**].

Capitolo 6

Estensione di ODB-Tools per introdurre regole attive

6.1 Regole attive

Si cerca di estendere la grammatica ODL per permettere la definizione di eventi e di regole ECA. Non potendo fare riferimento a nessun standard si è deciso di utilizzare una grammatica che permetta l'implementazione di tutte le caratteristiche delle regole attive e che sia il più intuitiva possibile.
Per mantenere la compatibilità con gli schemi precedenti, le modifiche della grammatica non toccano la definizione delle regole "vecchie" denominate rule. Cosí si dà la possibilità all'utente sia di definire vincoli di integrità senza evento sia di definire regole attive.
Come già spiegato in precedenza il paradigma delle regole attive ECA prevede una parte evento, una parte condizione ed una parte azione. La sintassi di una regola attiva sarà dunque:

Definizione 1 *< RuleDcl >:*

< RuleDcl > ::= **ecarule** *< Identifier >* **on**(*< EventName >* | *< SysEvent >*)
if *< Condition >* **do** *< Action >* [*< Option >*]

La parola chiave **ecarule** identifica la definizione della regola attiva (distinguendosi dalla parola chiave **rule** che identifica un semplice vincolo d'integrità) e precede il nome della regola stessa. La parola chiave **on** precede la parte evento, la parola **if** prelude alla condizione e, infine, la parola **do** precede l'azione.

Tutta la parte *option* è opzionale e verrà meglio discussa più avanti.
Prendiamo in esame le varie parti per discuterne più in dettaglio le caratteristiche facendo riferimento allo schema CLINIC riportato qui sotto in ODL di ODMG93.
In tutto il capitolo si fa riferimento a classi ed a oggetti ma le argomentazioni sono valide anche per sistemi relazionali con tabelle e tuple.

6.1.1 Lo schema di esempio: CLINIC

```
interface Person (extent persons  key number_card)
{       attribute string name;
        attribute struct Address_s
                        {
                        string street;
                        string city;
                        string tel_number;
                        } address;
        relationship set<Request> requested
                inverse Request::by;
        relationship Clinical_folder have
                inverse Clinical_folder::of;
        attribute integer number_card;
        attribute string state;
        unsigned short age();
        void print();
        void file();
};
interface Clinical_folder (extent clinical_folders  key number)
{
        attribute unsigned short number;
        attribute string last_update;
        relationship Person of
                inverse Person::have;
        relationship set<Visit> with
                inverse Visit::inc;
        void send(in string address);
        bollean ask();
};

interface Visit (extent visits  key number)
{
        attribute integer number;
```

```
        attribute string type;
        relationship Doctor doct
                inverse Doctor::doc;
        relationship Clinical_folder inc
                inverse Clinical_folder::with;
        attribute string result;
        attribute string state;
        attribute integer ticket;
        void print ();
        string assign(in string type);
        integer cl_ticket(in integer number);
        boolean n_esente(in integer n_c);
        boolean n_payed();
        string today();
        void show();
};

interface Doctor  (extent doctors)
{
        attribute string name;
        attribute Address_s address;
        attribute string specialization;
        relationship set<Visit> doc
                inverse Visit::doct;
        void salary();
};

interface Missed : Visit
{
        attribute string motivation;
};

interface Request : Visit   (extent requests)
{
        attribute string date_req;
        attribute string date_fixed;
        relationship Person by
                inverse Person::requested;
        relationship Room in_the
                inverse Room::req;
        void print(in integer num_copies);
};
```

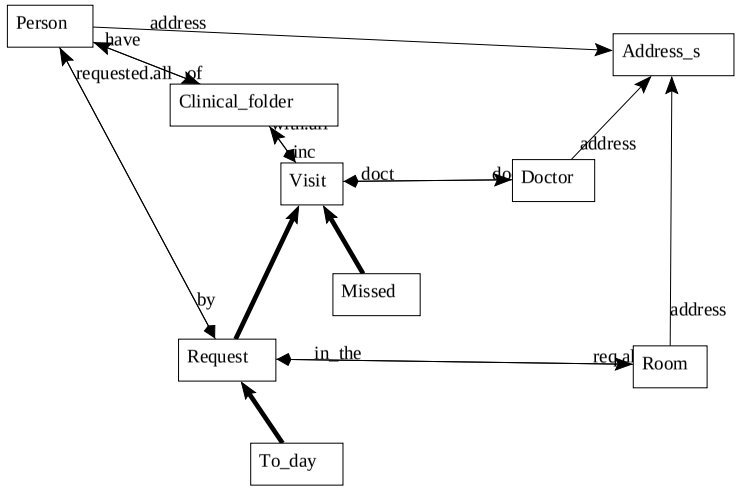



Figura 6.1: Schema dell'esempio in ODB-Tools

```
interface To_day : Request
{
};

interface Room
{
        attribute Address_s address;
        attribute unsigned short number;
        relationship set<Request> req
                inverse Request::in_the;
};
```

Nella figura 6.1.1 viene illustrato in veste grafica lo schema qui sopra.

6.1.2 Evento

Esistono due categorie di eventi principali, evento utente ed evento sistema. Quindi avremo:

- $< SysEvent >$ Eventi di sistema che sono scatenati da azioni di modifica fatte dall'utente nella gestione del DB, come insert, update e delete. Questa tipologia di eventi viene rilevata da tutti i DBMS commerciali. La sintassi per definire questi eventi è la seguente.

  Definizione 2 $< SysEvent >$*:*

  $< SysEvent > \ ::= \ (\textbf{insert} \mid \textbf{update} \mid \textbf{delete})\ \textbf{in}\ < ClassIdentifier >$

  Dove le parole chiave **insert**,**update** e **delete** indicano l'azione di modifica che scatena la regola, mentre $< ClassIdentifier >$ indica il nome della classe a cui è associata la regola.

  Esempio 9 *Se vogliamo che una regola venga attivata ad ogni inserimento nella classe Request non facciamo altro che definire la regola in questo modo:*

  ```
  ecarule rule1 on insert in Request if ...
  ```

- $< EventName >$ Eventi definiti dall'utente. Quando l'utente definisce un evento è obbligato ad assegnargli un nome. Ed è proprio il nome che viene richiamato nella definizione delle regole $< EventName >$.

Quindi un utente che vuole utilizzare un evento diverso da quelli di sistema, dovrà prima di tutto definirlo, e poi riprenderlo nella regola. Per definire un evento si deve seguire la sintassi specificata qui sotto:

Definizione 3 $< EventDcl >$*:*

$$
\begin{aligned}
< EventDcl > \ &::= \ \textbf{define event} \ < EventName > \ < EventDef > \\
< EventDef > \ &::= \ < MethodEvent > \mid < TimeEvent > \\
< EventName > \ &::= \ < Identifier >
\end{aligned}
$$

define event sono le parole chiave che precedono il nome $< EventName >$ dell'evento e la sua definizione $< EventDef >$. Nella implementazione attuale l'utente può definire solo due tipologie di evento, anche se la grammatica definita è facile da estendere. In futuro, quindi, sarà possibile inserire altre tipologie di evento.

- $< MethodEvent >$:

  Definizione 4 $< MethodEvent >$

  $$
  \begin{aligned}
  < MethodEvent > \ ::= \ & (\textbf{before} \mid \textbf{after}) \\
  & < ClassIdentifier > . < FunctionDef >
  \end{aligned}
  $$

  before ed **after** sono le due parole chiave che indicano quando si deve scatenare la regola rispettivamente prima o dopo l'esecuzione del metodo, $< ClassIdentifer >$ indica la classe di appartenenza del metodo che ha come nome quello specificato in $< FunctionDef >$, la cui sintassi verrà precisata più avanti. Nel caso peró che un metodo sia overloaded si possono anche specificare i parametri per evitare confusione. Vediamo qualche esempio per chiarire meglio:

  Esempio 10 *La classe Person dell'esempio ha il metodo con signature void print(). La definizione di un evento che viene segnalato prima dell'esecuzione dell'operazione è la seguente:*

  ```
  define event e1 before Person.print();
  ```

  La classe Doctor ha il metodo chiamato salary() che permette di cambiare lo stipendio. Se si vogliono fare controlli dopo che lo stipendio è stato cambiato allora si può definire un evento di questo tipo:

  ```
  define event e2 after Doctor.salary();
  ```

  Il metodo, specificato nell'evento definito dall'utente, può essere direttamente presente nella definizione della classe, *ClassIdentifier*, oppure può essere stato ereditato da una superclasse. Inoltre non è importante il tipo di ritorno del metodo, per questo non compare nella definizione dell'evento.

- $< TimeEvent >$:

  Definizione 5 $< TimeEvent >$*:*

  $$
  \begin{aligned}
  < TimeEvent > \ ::= \ & \textbf{every} \ < IntegerLiteral > \ < Cadency > \\
  & [\textbf{at} \ (< IntegerLiteral > : < IntegerLiteral > \mid \\
  & < IntegerLiteral >, < IntegerLiteral >)] \\
  < Cadency > \ ::= \ & \textbf{year} \mid \textbf{month} \mid \textbf{day} \mid \textbf{hour} \mid \textbf{minute}
  \end{aligned}
  $$

every precede un numero che, insieme a *Cadency*, indica la frequenza della segnalazione dell'evento. Opzionale, si può anche inserire l'ora in cui la regola deve essere attivata (Ore:Minuti) o la data (Giorno,Mese) a partire dal quale viene segnalato.

Esempio 11 *Si vuole definire una regola che venga attivata ogni cinque giorni alle ore 20:00. Allora la ecarule dovrà fare riferimento all'evento seguente:*

```
define event t1 every 5 day at 20:00;
```

Oppure si può attivare una regola ogni due mesi, allora l'evento diventa:

```
define event t2 every 2 month;
```

Infine in giorni particolari dell'anno:

```
define event t3 every 1 year at 6,11
```

Nei $< SysEvent >$ e nei $< MethodEvent >$ si fa riferimento implicito a degli oggetti. Come verrà spiegato nel paragrafo successivo, quando viene segnalato un $< MethodEvent >$ si sottintende che ad un oggetto viene fatta richiesta di eseguire un metodo, mentre quando viene segnalato un time event nessun oggetto viene sottinteso.

6.1.3 Condizione

La parte condizione permette all'utente di verificare particolari condizioni prima di eseguire l'azione specificata nella regola. Se la condizione non è verificata, cioé risulta falsa, l'azione non viene eseguita. La sintassi è:

Definizione 6 *< Condition >:*

< *Condition* > ::= **true** | (< *Query* >) < *RuleConstOp* >
< *LiteralValue* > | < *RuleBody* > |
< *ClassIdentifier* > . < *FunctionDef* >

< *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* > **)** |
< *Identifier* > ()

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

La condizione mi permette anche di reperire gli oggetti con i quali eseguire l'azione. Gli oggetti vengono "identificati" da degli iteratori, che possono

essere generali o speciali, concetto che risulterà chiaro più avanti. Infatti per alcune operazione posso utilizzare iteratori generali (X, S, ecc.), per altre operazioni invece devo utilizzare iteratori speciali (O, NEW, OLD). Quando un utente definisce una regola che viene attivata da una modifica (update) del database, allora nella regola si può far riferimento all'oggetto precedente la modifica OLD, oppure successivo alla modifica NEW. Quindi per chiarire meglio:

- **NEW** fa riferimento ad un oggetto appena inserito o appena modificato.
- **OLD** fa riferimento alla vecchia copia dell'oggetto appena modificato oppure si riferisce ad un oggetto appena cancellato.
- **O** fa riferimento all'oggetto a cui è stata richiesta l'esecuzione di un metodo.

Ogni tipologia di evento fa riferimento ai suoi iteratori speciali: per gli eventi di sistema sono validi gli iteratori che implementano la *history* del database, NEW e OLD, per i method event è valido l'iteratore O e per gli eventi tempo si devono utilizzare gli iteratori generali.

Esempio 12 *Facendo riferimento agli eventi definiti in precedenza, si vuole una regola che verifichi, prima della stampa, se il dottore si chiama "Carlo", nel qual caso aggiorni la locazione (Room) a "3".*

```
ecarule rule1 on e1 if O.doc.name=''Carlo''
  do O.in_the.number=''3'';
```

Oppure, se un paziente si trasferisce, si vuole archiviare la sua cartelle clinica.

```
ecarule rule2 on update on Person if NEW.state=''Transfer''
   do NEW.file();
```

La definizione della condizione viene considerata obbligatoria, però in alcuni casi potrebbe essere inutile. Per risolvere il problema si è prevista la possibilità di inserire anche la parole chiave **true**, imponendo così sempre condizione vera. Però se usato in combinazione con gli eventi tempo viene segnalato errore, infatti in questo caso, nella parte evento, non viene reperito nessun oggetto su cui lavorare nell'azione.

Esempio 13 *Questa regola è sbagliata. L'azione non può far riferimento a nessun oggetto. Infatti sia nell'evento che nella condizione non si reperisce nessun oggetto .*


```
ecarule rule3 on t1 if true do ....
```

La condizione accetta anche l'esecuzione di un metodo, ovviamente il tipo di ritorno del metodo deve essere un boolean altrimenti viene segnalato errore. Una richiesta di questo tipo può essere fatta solo con $< MethodEvent >$ e $< SysEvent >$. Infatti solo con questi si fa riferimento diretto a degli oggetti, con un evento tempo invece non si fa riferimento a nessun oggetto.

Esempio 14 *Per vedere se il paziente è esente dal ticket si utilizza il metodo esente(in integer num_card), che accetta in ingresso il numero della tessera sanitaria.*

```
ecarule rule4 on insert in Request
   if NEW.esente(NEW.by.number_card)
     do NEW.ticket=0;
```

Vengono inoltre accettate anche diverse condizioni di appartenenza, presenti nella sintassi di $< RuleBody >$, che ora vengono elencate. Condizioni di appartenenza ad una classe, ci si chiede cioè se un oggetto rientra nell'estensione di una classe. Condizioni sul tipo di attributo, cioè si verifica se un attributo rientra in un particolare tipo. Condizioni sul valore di un attributo, cioè si verifica se l'attributo assume un particolare valore.

Esempio 15 *Nel caso seguente ci si chiede se l'oggetto appena inserito, quindi identificato dall'iteratore speciale NEW, fa parte della classe Request:*

```
NEW in Request
```

In questo caso invece ci si chiede se l'età della persona identificata con l'iteratore speciale O, è compresa tra 19 e 24:

```
O.age in range {19, 24}
```

Possiamo anche verificare l'uguaglianza di un attributo con il valore di ritorno di un metodo:

```
X.data_fixed= string today()
```

In ultimo consideriamo il caso in cui il nome della persona sia uguale ad Andrea:

```
NEW.name=''Andrea''
```

Se si devono modificare più oggetti di una classe, si possono utilizzare i costrutti *forall* ed *exists*. Tenendo presente la sintassi di questi due costrutti, si nota come sia necessario un iteratore $< Identifier >$ che identifichi gli oggetti della classe presa in considerazione, $< DottedName >$. Gli oggetti devono rispettare determinate condizioni che vengono specificate dopo i ":". Nel primo caso $< Forall >$, vengono reperiti tutti e soli gli oggetti che soddisfano le condizioni finali (specificate dopo i ";"), nel secondo caso $< exists >$ invece vengono reperiti tutti gli oggetti della classe se e solo se almeno un oggetto tra questi rispetta le condizioni.

Esempio 16 *Supponiamo di voler stampare ogni anno tutte le visite eseguite dal dottor Carlo*[1].

```
ecarule Carlo on t3 if forall X in Visit :
  X.doc.name="Carlo"
     do X.print(1);
```

In ultimo, il costrutto *select-from-where* permette di verificare il numero degli oggetti che rispondono a certe condizioni. Nell'implementazione attuale, però, la condizione che utilizza il costrutto *select-from-where* non reperisce nessun oggetto. Quindi nella parte azione si deve far riferimento agli oggetti sottintesi nella parte evento. Non viene accettata, dunque, la combinazione $< TimeEvent >$ e $< Query >$ poichè non viene reperito nessun oggetto.

Esempio 17 *Supponiamo di voler imporre un limite massimo di richieste, per esempio 100:*

```
ecarule max on insert in Request
        if (select * from Request )>100
            do abort;
```

Ad ogni inserimento si verifica che il numero degli oggetti della classe Request sia inferiore a 100. Se ciò non accade si effettua l'abort dell'inserimento.

6.1.4 Azione

La parte azione specifica tutte le operazioni che devono essere eseguite se la condizione viene verificata. Innanzi tutto si deve verificare che l'iteratore con cui si indicano le variabili o metodi sia lo stesso utilizzato nella parte

[1] Il parametro 1 del metodo *print()* indica il numero di copie di stampa

condizione o sotto inteso nella parte evento. La sintassi che deve essere rispettata è:

< Action > ::= < BodyList > | < Identifier > . < FunctionDef > |
< Identifier > . < FunctionDef > **AND** < Action > |
< BodyList > **AND** < Action > |
< **abort** > | < Identifier > **.delete**
< DottedName > = < DottedName > < EcaOp >
< LiteralValue >
< EcaOp > ::= (+ | − | ∗ | /)

Parte della sintassi è già stata trattata in precedenza negli esempi, è comunque opportuno sottolineare cosa l'azione può eseguire:

- Esecuzione di un metodo. Già negli esempi precedenti si è più volte mostrata questa possibilità.
- Cambiamento di valore di un attributo. In questo caso è possibile modificare il valore di un attributo sia direttamente oppure grazie al calcolo di un metodo.

  Esempio 18 .

  ```
  X.salary = ''100000000''
  X.ticket = integer cl_ticket();
  ```

- Cambiamento di valore di un attributo tramite un'espressione. In questo caso non è necessario fare ricorso ad un metodo per calcolarsi il nuovo valore dell'attributo.

  Esempio 19 *Qui sotto possiamo vedere una incremento di una variabile contatore e un aumento del salario del 10*

  ```
  X.count = x.count + 1;
  X.salary = X.salary * 1.1;
  ```

- **abort**. Con questa istruzione si impedisce alla transazione di eseguire il commit. La grammatica dà anche la possibilità di usare questo comando in AND con gli altri, però dal punto di vista sintattico questo non ha molto senso.

- **delete.** La parola si spiega da sola, tutti gli oggetti presi in considerazione vengono cancellati dalla classe di appartenenza. Anche in questo caso si può utilizzare questo comando in AND con gli altri anche se va posto solo in ultima posizione.

6.1.5 Parte Opzioni

Si vuole inoltre dare la possibilità all'utente di specificare delle opzioni per controllare meglio il comportamento delle regole. ODB-Tools viene utilizzato a livello di progettazione, cioè viene costruito uno schema, controllato ed implementato su un database di appoggio (Per ora è in via di sviluppo solo per UNISQL).
Lo scopo di ODB-Tools non è però quello di legarsi ad un database fisico, bensì di offrire all'utente un potente strumento di ottimizzazione di uno schema. E' per questo motivo che sono state inserite nelle opzioni tutti i parametri che possono essere utili nel controllo del comportamento delle regole attive, anche se , praticamente, nessun DBMS commerciale li implementa tutti.
La grammatica è stata estesa cercando ancora di mantenere il significato intuitivo.

< OptionList > ::= < Option > | < Option >, < OptionList >
< Option > ::= **granularity** = < GranOpt > |
EC = < CouplingOpt > |
CA = < CouplingOpt > |
consumption_scope = < ConsScopeOpt > |
consumption_time = < ConsTimeOpt > |
atomic = < AtomicOpt > |
precedes (< NameList >) |
follows (< NameList >)
< GranOpt > ::= **instance** | **set**
< CouplingOpt > ::= **immediate** | **delayed** | **deferred**
< ConsScopeOpt > ::= **local** | **global** | **no**
< ConsTimeOpt > ::= **condition** | **execution**
< AtomicOpt > ::= **atomic** | **no**
< NameList > ::= < Identifier > | < Identifier >, < NameList >

Le opzioni sono esattamente quelle citate nel primo capitolo di questa tesi, ma per maggior chiarezza ne richiamo brevemente il loro significato. Granularity indica se una regola deve reagire ad ogni modifica ad un oggetto (instance oriented) o una volta solo alla modifica di un gruppo di oggetti (set oriented). Coupling Mode sincronizzano evento-condizione (EC) e condizione-azione (AC), possono assumere i valori di immediate immediato, delayed spostato nel tempo e deferred subito prima del commit. Atomic indica se la regola può essere interrotta oppure viene eseguita in modo atomico, può assumere i valori atomic o no. Consumption_Scope indica l'interazione tra le regole e può assumere i valori local, global e no. Consumption_Time indica quando avviene il consumo dell'evento a livello di condizione (condition) o di azione (execution). Precedes e Follows indicano l'ordine di esecuzione delle regole, nel primo vengono messe le regole che devono precedere quella in considerazione, nel secondo inserisco le regole che devono essere posposte.

Esempio 20 *La regola attiva seguente mostra come devono essere inserite le opzioni di comportamento. Si noti come la granularità sia definita per ogni istanza e che la regola debba essere eseguita dopo la regola attiva* ticket

```
ecarule print on insert in Request if true
    do NEW.print(1)
    granularity = instance, follows (ticket);
```

6.1.6 Lo schema CLINIC con regole attive

Con alcuni esempi mostro come è possibile scrivere delle regole attive, tenendo sempre presente l'esempio della Clinica.
Se un paziente si trasferisce in un altro stato si vuole archiviare la sua cartella clinica e le sue visite, allora si può scrivere:

```
ecarule archi on update in Person if NEW.state=''transfer''
    do NEW.file();
```

Ogni volta che la classe Person subisce un update la regola viene attivata, viene verificato l'attributo *state* dell'oggetto appena modificato e se corrisponde a "transfer" viene lanciato il metodo *file()* che archivia tutto quello che riguarda quell'oggetto.
Si vuole che ogni mattina tutte le visite del giorno vengano inserite nella classe To_day.

```
define event morning every 1 day at 8:00;
ecarule start on morning if forall X in Request :
                            X.date_fixed= string today()
    do X in To_day;
```

In questo caso si definisce un *Time Event* che attivi la regola ogni giorno alle 8:00 del mattino. Si collega la nostra regola all'evento appena definito e, quando la regola viene attivata, si reperiscono tutti gli oggetti in Request che hanno l'attributo *date_fixed* uguale al valore di ritorno del metodo *today()*. Tutti gli oggetti reperiti vengono spostati nella classe To_day.
Lo stesso ragionamento può essere fatto per cancellare tutti gli appuntamenti dalla classe To_day alla fine della giornata.

```
define event evening every 1 day at 21:00;
ecarule finish on evening if forall X in Request :
                            X.state=''execute''
    do X.delete;
ecarule finish1 on evening if forall X in Request :
                             X.state=''not_execute''
    do X in Missed and X.delete;
```

In questo caso si definisce un *Time Event* che viene segnalato tutte le sere alle 21:00. Le regole collegate a questo evento vengono attivate. La prima cancella dalla classe Request tutti gli oggetti che hanno l'attributo *state* uguale ad "execute" cioè tutte le visite eseguite quel giorno. Per l'ereditarietà anche la classe To_day eredita questa regola, quindi anche in questa classe vengono cancellate tutte le visite eseguite. Nella seconda regola, tutte le visite che non sono state eseguite vengono inserite nella classe Missed e poi cancellate dalla classe Request.
Si vuole calcolare in modo automatico il ticket da pagare per una visita specialistica.

```
ecarule ticket on insert in Request if
                        NEW.n_esente(NEW.by.number_card)
    do NEW.ticket= integer cl_ticket(NEW.by.number_card)
    granularity=instance;
```

Ad ogni inserimento nella classe Request, se il richiedente non è esente da ticket, si aggiorna il campo *ticket* eseguendo il metodo *cl_ticket* che ha come parametro il numero della previdenza sociale del richiedente.
Inoltre può essere utile stampare l'appuntamento appena fissato però per evitare che nella stampa non compaia il ticket si deve far seguire la regola seguente a quella soprascritta.

```
ecarule print on insert in Request if true
    do NEW.print(1)
    granularity=instance, follows (ticket);
```

Ad ogni inserimento nella classe Request viene stampata una copia dell'oggetto appena inserito dopo aver eseguito la regola *ticket*. Si può ottenere lo stesso risultato anche non utilizzando l'opzione *follows*.


```
define event a_ticket after Request.ticket();
ecarule print on a_ticket if true
    do O.print(1);
```

In questo caso, prima si definisce un *Method Event* che segnala l'evento dopo l'esecuzione del metodo *ticket()*, poi si definisce una regola che, quando attivata, stampa una copia della richiesta.
Se si vuole vedere l'esito della visita si deve controllare che il ticket sia stato pagato.

```
define event show before Visit.show();
ecarule eca_show on show if O.n_payed()
    do abort;
```

Viene definito un *Method Event* che segnala un evento prima della esecuzione del metodo *show* che mostra l'esito della visita. A questo punto si definisce una regola che si attiva con questo evento e che verifica che il ticket sia stato pagato eseguendo il metodo *n_payed* che ritorna vero se non è stato pagato nulla. La parte azione annulla tutte le operazioni e quindi anche l'esecuzione del metodo *show*, infatti, se non è ancora stato pagato il ticket, non si ha diritto a vedere l'esito dell'esame.
Di seguito vengono riportate tutte le regole regole attive inserite nello schema di esempio CLINIC e nella figura 6.2 viene riportata la forma grafica mostrata dell'applet **scvisual**.

```
define event show before Visit.show();
ecarule show_eca on show if O.n_payed()
       do abort;
ecarule pr_request on insert in Request if true
       do NEW.print1(1);
define event morning every 1 day at 8:00;
ecarule start on morning if forall X in Request :
     X.date_fixed = string today()
       do X in To_day;
define event evening every 1 day at 21:00;
ecarule finish on evening if forall X in Request :
     X.state="executed"
       do X.delete;
ecarule finish1 on evening if forall X in Request :
     X.state = "not_executed"
       do X in Missed and X.delete;
ecarule archi on update in Person if NEW.state="transfert"
       do NEW.file1() and NEW.delete;
ecarule archi1 on delete in Clinical_folder
     if OLD.of.state = "alive"
       do abort;
```

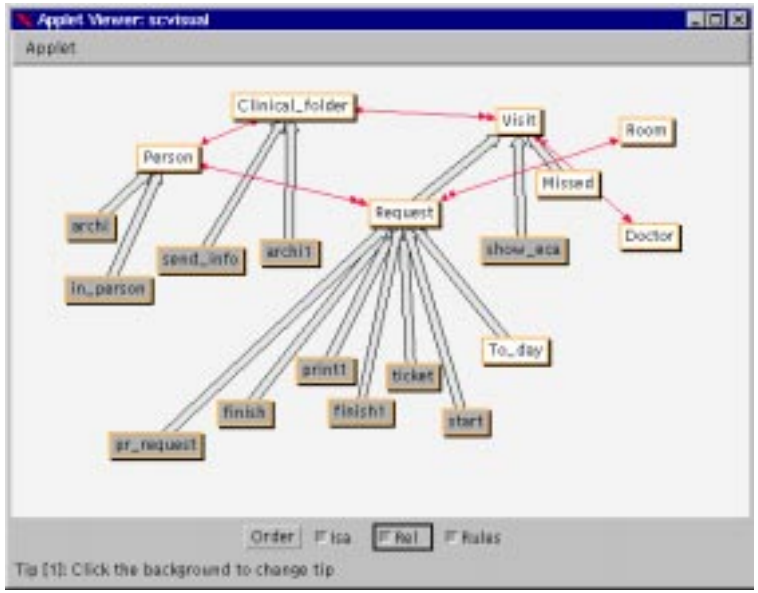


Figura 6.2: Schema CLINIC con regole attive

```
ecarule ticket on insert in Request
     if NEW.n_esente(NEW.by.number_card)
        do NEW.ticket=integer cl_ticket(NEW.by.number_card,
             NEW.by.number_card)
         granularity=instance;
ecarule print1 on insert in Request if true
        do NEW.print1(1)
         granularity=instance, follows(ticket);
define event b_send before Clinical_folder.send();
ecarule send_info on b_send if O.general_status="private"
        do O.ask();
```


6.2 Integrazioni Future

L'implementazione delle regole attive fatta in questa tesi è solo un punto di partenza. Infatti per gestire dinamicamente database complessi sono necessarie alcune aggiunte.
Per esempio sarebbe utile la possibilità di definire eventi composti con operatori OR, AND, ecc. e a questo scopo ampliare anche il numero degli eventi definibili dall'utente.
Per quanto riguarda la parte condizione attualmente è possibile utilizzare l'AND solo se la condizione è introdotta dal costrutto $< Forall >$, in futuro si potrebbero aggiungere gli operatori OR, AND e NOT per poter comporre condizioni diverse .
Nella parte azione, invece, potrebbe essere utile la possibilità di attivare o disattivare le altre regole attive a seconda delle condizioni in cui si trova il database.
In ultimo si potrebbe aggiungere una alternativa alle azioni, da eseguire quando la parte condizione non viene verificata.

Capitolo 7

Implementazione delle regole attive in ODB-Tools

7.1 ODL_Trasl : estensione del traduttore per l'interpretazione di regole attive

7.1.1 Struttura del programma

La sintassi di ODL è stata diffusa in formato *lex&yacc*[1], tale scelta è in accordo con la politica dell'ODMG di rendere ODL facile da implementare.
Attorno alla sintassi Lex&Yacc sono state aggiunte:

1. Le *actions* della parte Yacc. Nelle actions vengono memorizzati tutti i concetti letti dall'ODL, vengono inoltre eseguiti alcuni controlli semantici.
2. Routine di controllo della coerenza dei dati. Tra queste routine è presente l'algoritmo di traduzione delle rules.
3. Routine di stampa in formato OCDL.

Il programma è composto dai seguenti moduli

- modulo principale (`c_main.c`):
  Si ha l'inizializzazione delle variabili globali, e si chiamano in sequenza: *il parser*, *il controllore coerenza*, *la routine di stampa dei concetti in OCDL*
- parser della sintassi ODL:
  È composto dal modulo *lex* (`odl.l`) e dal modulo *yacc* (`odl.y`). Svolge il controllo sintattico della sintassi ODL grazie alle routine generate con Lex&Yacc. Durante tale controllo riempie le strutture dati in memoria.

[1] si veda appendice B

- controllore di coerenza (`c_coeren.c`):
  Controlla l'univocità dei nomi di: `const`, `struct`, `typedef`, `interfaces`, `operation`, `rule` e `regole attive`. Controlla che esistano i campi inversi delle "relazioni" e calcola la traduzione OCDL delle rules. Inoltre controlla che tutte le regole attive si riferiscano ad un evento definito.
- routine di stampa dei concetti in OCDL (`c_procdl.c`):
  Dalle strutture dati allocate dal parser estrae la forma OCDL.
- algoritmo di terminazione (`s_terminator.c`) Esamina la proprietà di terminazione del set di regole attive definite nello schema. L'algoritmo di terminazione sarà spiegato in tutte le sue parti nel prossimo capitolo 8.

Di seguito saranno descritte solo le parti del programma interessate alla estensione delle operazioni. Per eventuali chiarimenti sulla struttura originale del traduttore si veda la Tesi di laurea di A. Corni [?] e la tesi di laurea si S. Riccio [?].

7.1.2 Le strutture dati

Viene presentata a grandi linee la struttura con cui è memorizzato uno schema, inoltre è presente una accurata descrizione delle strutture che conservano informazioni relative agli eventi e alle regole attive.

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

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

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

Il campo `prop` punta alla lista delle proprietà.
Il campo `next` serve per la gestione della lista globale di tutte le `interface` dello schema.

Rappresentazione degli *eventi* Gli eventi definiti dall'utente (*Method Event* *Time Event* si veda 6.1.2) sono memorizzati nella lista degli eventi `Event_type`.

```
 struct s_event_type
    {
     char *name ;       /*nome dell'evento*/
     struct s_event_def_list *e_def;
     /* struttura che punta alla definizione dell'evento*/
     struct s_event_type *next;
    }
 *Event_type;
```

dove `name` mantiene il nome dell'evento mentre `e_def` è un puntatore alla definizione dell'evento.
La struttura seguente permette di memorizzare tutte le definizioni degli eventi, anche dei *System Event* che sono definiti direttamente nel corpo delle regole, si veda sempre 6.1.2.

```
struct s_event_def_list
   {
     char type ;        /*Flag puo' assumere i valori
                         * 'm' = method event -- evento metodo
                         * 't' = time event -- evento tempo
                         * 's' = system event -- evento sistema
                         */
     union
     {
      struct
      {
       char tempo;      /* flag, che identifica il tempo di
```



```
                         * segnalazione dell'evento, puo' essere
                         * 'b' = before
                         * 'a' = after
                         */
       char *classname;   /* nome variabile interessata */
       struct s_operation_param *opname;
       /* si riferisce alla signature del metodo chiamato */
      } m_e;
     struct
      {
       char *freq;   /* memorizza la frequenza nel time event*/
       char *tipo;  /* flag puo' essere
                     * 'y' = year -- anno
                     * 'm' = mese -- month
                     * 'd' = day -- giorno
                     * 'h' = hour -- ora
                     * 'n' = minute -- minuto
                     */
       char ot;     /* flag puo' essere
                     * h = ora
                     * d = data
                     */
       char *h_time;  /* ore del tempo o giorno di data*/
       char *n_time;  /* minuti del tempo o mese di data*/
      } t_e;
     struct
      {
       char mode;   /* puo' assumere i vcalori
                     * 'i' = insert
                     * 'd' = delete
                     * 'u' = update
                     */
       char *classname;
      } s_e;
     } ee;
    struct s_event_def_list *next;
   }
 *Event_def_list;
```

I campi della struttura hanno il seguente significato:

- **type** indica la tipologia dell'evento memorizzato nel record.
- **ee** è una union che a seconda del valore di **type** permette di utilizzare le strutture al suo interno.

m_e per i *Method Event*

t_e per i *Time Event*

s_e per i *System Event*

All'interno di ogni struttura ci sono i campi necessari per memorizzare tutto ciò che prevede la grammatica.

- **next** questo campo è stato inserito in previsione di una estensione delle regole permettendo così di inserire eventi composti che nell'implementazione attuale non sono previsti.

Le operazioni nei *Method Event* vengono memorizzate grazie alla struttura chiamata `Operation_param`. I record di questa struttura permettono di memorizzare completamente la signature dell'operazione.

```
struct s_operation_param
{
 char   type;         /* tipo di parametro */
 union
 {
  char   *OperationName;
  char   *DottedName;
  char   *Value;
 } r;
 char *attribute_param;// in o out o inout
 char *param_type; //se 'n' e' il tipo di ritorno
 struct s_declarator_type *ParamDeclarator;/* e' il "nome" */
 struct s_operation_param *next;
} *Operation_param;
```

La struttura ha un duplice utilizzo:

- permette di memorizzare le operazioni dichiarate nella interface delle classi
- permette di memorizzare le funzioni dichiarate nelle rule

Descrizione dei campi della struttura :

- **type** indica il tipo di parametro
  - se type è uguale a 'n' allora si è nel primo elemento della lista dei parametri, quindi si memorizza:

    - il nome della operazione in **OperationName**
    - il tipo di ritorno in **paramtype**
  - se type è uguale a 'p' allora si è dichiarata l'operazione all'interno di una interface di una classe, in tal caso si memorizza:
    - il nome del parametro in **ParamDeclarator**
    - l'attributo del parametro in **attribute_param**
    - il tipo del parametro **param_type**
  - se type è uguale a 'd' si è dichiarato l'operazione all'interno di una rule ed il parametro presenta un DottedName come valore di passaggio. In tal caso si memorizza:
    - l'attributo del parametro in **attribute_param**
    - il nome del dottedname in **DottedName**
    - il tipo del parametro in **param_type**. Il tipo del parametro è facoltativo, nel caso in cui non venga specificato esso viene calcolato dal programma.
    - il nome del parametro in **ParamDeclarator**. Nel caso in cui l'operazione venga dichiarata in una rule, nella sintassi ODL non appare il nome del parametro formale, ma solamente il nome del parametro attuale, il quale può essere un dottedname. In questo caso il programma assegna un nome al parametro formale e lo assegna alla variabile ParamDeclarator. Tale nome viene generato in modo molto semplice, inizia con "param" seguito da un numero progressivo che identifica ogni parametro (ad esempio param1, param2 ecc...)
  - se type è uguale a 'v' si è dichiarata l'operazione all'interno di una rule ed il parametro presenta una costante come valore di passaggio.In tal caso si memorizza:
    - l'attributo del parametro in **attribute_param**
    - il valore in **Value**
    - il tipo del parametro in **param_type**. Anche in questo caso il tipo può essere ricavato dal programma.

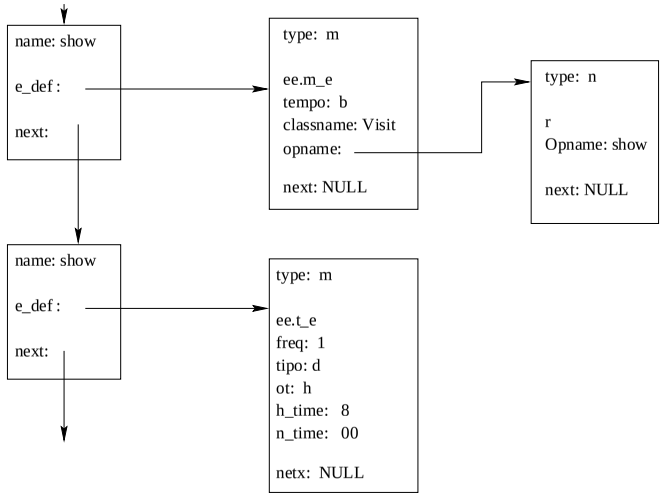


Figura 7.1: Struttura in memoria degli eventi

  * il nome del parametro in **ParamDeclarator**. Anche in questo caso il nome del parametro viene determinato dal programma.

- **next** il puntatore al prossimo elemento della lista

Per una descrizione più dettagliata sulla memorizzazione delle operazioni si veda [?].

Esempio di rappresentazione degli eventi Vediamo graficamente un esempio di come vengono rappresentati gli eventi definiti dall'utente. Prendo come riferimento l'esempio CLINIC riportato nel 6.1.1:

```
define event show before Visit_show();
define event morning every 1 day at 8:00;
```

Nella 7.1.2 si notano sulla sinistra i record della struttura `s_event_type`, mentre sulla destra i record della struttura `s_event_list`.

Rappresentazione delle *regole attive* Si mostra ora in che modo le informazioni relative alle regole attive sono memorizzate, ovvero la rappresentazione interna delle informazioni delle regole attive lette in fase di parsing del formato ODL esteso.

```
struct s_ecarule_type
 {
   char *name;    /* nome dell'ecarule */
   char etype;    /* puo' assumere valori diversi
                   * 's' = sys event
                   * 'u' = user event
                   * con s significa che la regola e'
                   * legata a un evento del tipo insert,
                   * delete, update.
                   * se invece si ha un u significa che la
                   * regola e' legata a un nome di un evento
                   * definito interamente dall'utente.
                   * permette di selezionare il puntatore
                   * giusto nella union qui sotto
                   */
   union
    {
     char *eve;     /* nome dell'evento di triggering */
     struct s_event_def_list *el; /* puntatore alla
                                   * definizione del sys event
                                   */
    } gen;
   struct s_ecacond_list *cond;
   /*punta alla struttura che memorizza la condizione*/
   struct s_ecacond_list *act;
   /* punta alla struttura che memorizza l'azione*/
   struct s_eca_option_list *opt;
   /*punta alla struttura che memorizza le opzioni */
   struct s_ecarule_type *next;
 }
*EcaRule_type;
```

La struct `s_ecarule_type` serve per memorizzare le varie regole attive che vengono mantenute in una lista. Come si vede una regola attiva viene descritta da una parte event (**gen**), da una parte condizione (**cond**), da una parte azione (**act**) e da una parte opzione (**opt**). La parte evento può essere o un evento definito dall'utente, allora avrò memorizzato il nome dell'evento, o un *System Event* , allora avrò direttamente la definizione dell'evento. La parte condizione e la parte azione fanno riferimento ad una struttura dello stesso tipo, descritta qui sotto, che

permette di memorizzare tutte le varie tipologie esprimibili dalla grammatica. La perte opzione fa riferimento ad una struttura differente che memorizza in una lista tutte le opzioni definite insieme alla regole. Infine il campo **next** fa riferimento alla regola successiva.

Rappresentazione della *parte condizione* e della *parte azione* La struttura che segue permette di memorizzare tutte le possibili condizioni e azioni esprimibili dalla grammatica delle regole attive, vedi 6.1.3, 6.1.4.

```
struct s_ecacond_list
 {
  char type;    /* flag che puo' assumere i valori
                 * 't' -> true
                 * 'f' -> costrutto forall
                 * 'r' -> o costante o tipo
                 * 'e' -> chiamata esecuzione metodo
                 * 'a' -> abort (solo per azione)
                 * 'd' -> delete (solo per azione)
                 * 'o' -> operation (solo per azione)
                 * 'q' -> query (solo per condizione)
                 */
 union
  {
   char t;
   char *del;
   /* struct s_rule_body_list *f;*/
   struct s_rule_body_list *r;
   struct
    {
     char *classname;
     struct s_operation_param *o;
    } e;
   struct
    {
     struct s_query_list *q;
     char *operator;
     char *value;
     char *opty;
    } query;
   struct
    {
     char *first;  /* primo dottedname */
     char *second; /* secondo dottedname */
```



```
     char *op;     /* segno dell'operazione */
     char *value;  /* valore   */
    } o;
  } pp;
 char *ocdl;      /* memorizzo la tradizione in ocdl
                   * della condizione*/
 struct s_ecacond_list *next;
 char class[256]; /* serve per
                   * l'algoritmo di terminazione
                   */
}
*Ecacond_list;
```

La struttura ricalca quella costruita per gli eventi. Un campo **type** che discrimina la tipologia e una **union** che contiene tutti i campi per memorizzare in modo completo e corretto la definizione.
Le strutture e i campi presenti nella union sono associati con i valori nel type come nella mostrato nella tabella 7.1. Il campo **ocdl** memorizza la traduzione in OCDL della parte condizione. Questo campo viene riempito nel corso dell'esecuzione della parte di controllo di coerenza dove vengono tradotte in OCDL le condizioni che sono traducibili.
Anche il campo **class** viene riempito al momento dell'esecuzione del controllo di coerenza e mantiene il nome della classe, o delle classi a cui fa riferimento la parte in considerazione. Per la parte condizione questo campo viene utilizzato nel momento di scrittura della Visual Form, per la parte azione il campo è necessario per l'algoritmo di terminazione spiegato nel prossimo capitolo.
Vediamo ora più da vicino come vengono memorizzati casi denominati nella tabella 7.1 *forall, constante e tipo*.
In questo caso, siccome la sintassi grammaticale è identica a quella utilizzata nelle **rules** si è pensato di mantenere la stessa struttura che memorizza quest'ultime (Per maggiori informazioni vedi [**?**])

```
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
```

Tipologia	type	campo union	descrizione
true	t	-	-
forall, costante o tipo	r	struct r	è un puntatore alla struttura `s_rule_body_list`.
esecuzione metodo	e	struct e	il campo della struttura memorizza **classname** il nome della classe di appartenenza del metodo, il secondo, **o**, è un puntatore alla struttura `s_operation_param` che memorizza la signature del metodo.
query oql	q	struct query	la struttura query ha un puntatore alla struttura `s_query_list` che memorizza la query inserita dall'utente, un campo operator che memorizza l'operatore (=, <, >, ecc), un campo value che memorizza il valore a cui si deve comparare il risultato della query e un campo opty dove viene memorizzata la query ottimizzata da odbqo.
abort	a	-	-
delete	d	del	memorizza l'iteratore che identifica l'oggetto da cancellare
espressione	o	struct o	il campo **first** mantiene l'attributo da modificare, il secondo **second** memorizza l'attributo di riferimento, **op** il segno dell'operazione matematica da eseguire e **value** memorizza il valore numerico da utilizzare nella espressione.

Tabella 7.1: Correlazione tra il campo Type e la union


```
  * 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 condizioni
  *  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_operation_param *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 può valere:
  - ’c’ dichiarazione di costante
  - ’f’ la regola è un forall o un’exists
  - ’i’ dichiarazione di tipo
    In questo caso particolare, può 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_operation_param**).

Si può notare come un record di questa struttura è in grado di descrivere uno qualunque dei costrutti citati.

Rappresentazione della *parte opzione* In ultimo si riporta la struttura che memorizza la parte delle opzioni. Le informazioni che devono essere conservate sono il nome dell’opzione e il suo valore. Nel caso che l’opzione sia *follows* o *precedes* il valore viene sostituito dalla lista dei nomi delle regole attive che precedono o seguono quella in esame.


```
struct s_eca_option_list
 {
  char type;   /* flag che puo’ assumere diversi valori
                * g = granularity
                * e = ec coupling mode
                * a = ac coupling mode
                * s = consumption scope
                * t = consumption time
                * p = precedes
                * f = follows
                */
  union
   {
    char *value; /* valore delle opzioni
                  * i/o = instance oriented
                  * s/o = set oriented
                  * immediate
                  * delayed
                  * deferred
                  * condition
                  * execution
                  * global
                  * local
                  * no
                  */
    struct s_eca_name_list *list;
    /* contiene la lista delle precedenze */
   } oo;
  struct s_eca_option_list *next;
 }
*Eca_Option_List;
```

Nella figura 7.1.2 è riportata graficamente la memorizzazione della seguente regola attiva:

```
ecarule archi1 on delete in Clinical_Folder
  if OLD.of.state=’’alive’’
    do abort
     granularity = instance;
```

7.1.3 Descrizione delle funzioni

In questa tesi si fa riferimento solo alle nuove funzioni riguardanti le regole attive.

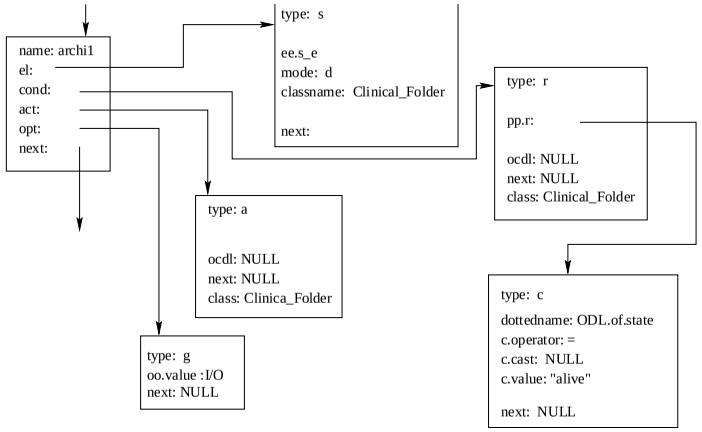


Figura 7.2: Struttura in memoria delle regole attive

Funzioni di gestione delle regole attive Di seguito si presentano le funzioni che utilizzano le strutture dati riservate alle regole attive e agli eventi per eseguire alcuni controlli di consistenza. Dal diagramma 7.3 è possibile vedere come è strutturato l'algoritmo di controllo semantico delle regole attive secondo la metodologia PHOS.
Segue la descrizione una ad una delle funzioni scritte nell'odl_trasl:

find_event permette di trovare l'evento referenziato nella regola.

ecarule_cond effettua il controllo della parte condizione della regola. Per ogni tipologia controlla che l'iteratore utilizzato sia corretto in riferimento all'evento inoltre verifica che gli attributi siano esistenti e definiti in modo corretto. Infine, nel caso di forall e di query, effettua la traduzione in OCDL.

ecarule_act effettua il controllo di coerenza della parte azione. Per ogni tipologia controlla che l'iteratore sia concorde con quello utilizzato nella parte condizione e nella parte evento, inoltre si verifica che gli attributi definiti siano coerenti con lo schema.

ecarule_opt effettua il controllo sulla parte opzione. Verifica in particolare che non ci siano definizioni di opzioni doppie e nel caso di *follows e precedes* verifica che i nomi nella lista di regole si riferiscano a regole effettivamente definite.

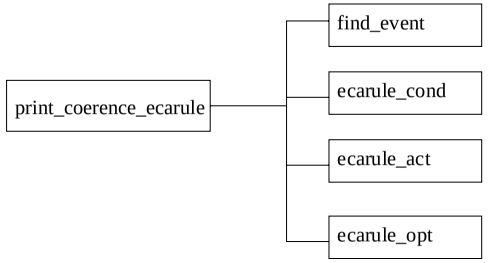


Figura 7.3: Diagramma PHOS del controllo di coerenza

optyrule completa il controllo di coerenza delle regole attive con l'ottimizzazione delle query eventualmente inserite. Infatti in presenza di query viene chiamato prima l'ocdl-designer, che scrive lo schema in forma cononica, e poi l'odbqo che ottimizza la query in riferimento allo schema canonico. La query ottimizzata viene memorizzata nella apposita struttura dati per poi sostituire quella originaria. Se la query non è ottimizzabile allora viene lasciata quella inserita dall'utente.

print_coerence_event effettua i controlli di coerenza sugli eventi definiti dall'utente. Controlla che gli eventi non facciano riferimento a classi o ad operazioni inesistenti.

Funzioni di scrittura delle regole attive Dopo aver effettuato tutti i controlli è necessario scrivere sul file di output lo schema tradotto in OCDL. Gli eventi e le regole attive non vengono tradotte completamente in OCDL, però è comunque necessario scriverle nel file di output.

print_eca effettua la scrittura sul file di output *nomeschema.sc* delle regole attive e degli eventi. Il formato di scrittura non è l'OCDL ma un formato che ricalca la struttura della definizione in ODL. Questo perchè non è necessario tradurre le regole in forma canonica.

print_vfform scrive il file di output *nomeschema.vf* che permette la visualizzazione dello schema in veste grafica grazie all'applet java **scvisual**.

7.2 OCDL_Designer

7.2.1 Le struttura dati

Lo schema viene memorizzato in cinque strutture a lista:

- **listaN** per i tipi valore e i tipi classe
- **listaB** per i tipi base
- **listaO** per le operazioni
- **listaE** per gli eventi
- **listaECA** per le regole

L'estensione del formalismo alle regole attive interessa solo due liste:

1. ListaE che contiene tutti gli eventi dello schema
2. ListaECA che contiene le regole attive

La lista ListaE Gli elementi della lista ListaE hanno la seguente struttura:

```
typedef struct lN{
char    *name;  /*nome di tipo o di classe o di evento*/
int     type;   /*costante che identifica il tipo */
int     new;    /*uguale a TRUE se e' un nuovo nome */
L_sigma *sigma; /*puntatore alla descrizione iniziale*/
L_sigma *iota;  /*puntatore alle descrizione trasformata*/
L_gs    *gs;    /*puntatore alla lista dei gs */
struct  lN *next;
} lN;
```

I campi, che interessano il lavoro di questa tesi, hanno i seguenti significati:

- name : stringa che descrive il nome dell'event
- type : costante che identifica il tipo, può assumere i seguenti valori:
  - T : tipo valore
  - C : classe primitiva
  - D : classe virtuale
  - RA_V o RA_T: antecedente di una regola
  - RC_V o RC_T: conseguente di una regola

  - OP : tipo operazione
  - EV : tipo evento
- sigma : puntatore alla lista che rappresenta la descrizione originale

Gli elementi della lista sigma hanno la seguente struttura:

```
typedef struct L_sigma{
char    *name;        /* nomi di tipi o di classi */
int     type;         /* costante che identifica il tipo */
void    *field;       /*puntatore per informazioni aggiuntive*/
struct L_sigma *next; /*punt. per rappresentare descrizioni annidate*/
struct L_sigma *and;  /*punt. per rappresentare congiunzioni */
char *attribute;      /*per conservare (in/out/inout) */
} L_sigma;
```

I campi hanno i seguenti significati:

- name : stringa che descrive il nome di tipo o di classe
- type : costante che identifica il tipo, nel caso degli eventi può assumere i seguenti valori:
  - EV_S : *System Event*
  - EV_M : *Method Event*
  - EV_T : *Time Event*
  - OP : tipo operazione
- field : puntatore a elementi di strutture contenenti informazioni aggiuntive
- next : puntatore per eventuali descrizioni annidate come nel caso dei *Method Event* per la definizione delle operazioni
- and : puntatore ad un elemento L_sigma utilizzato per rappresentare le intersezioni nelle descrizioni
- attribute : a seconda del tipo di evento memorizza informazioni sulla tipologia, per esempio *before*, *after*, *insert*, ecc.

Nella figura 7.4 viene mostrata la rappresentazione degli eventi presi come esempio in precedenza.

```
define event show before Visit.show();
define event morning every 1 day at 8:00;
```

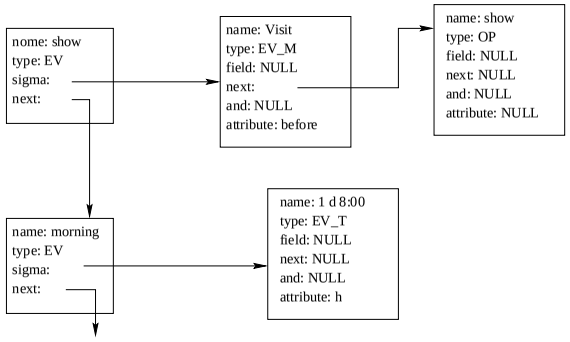


Figura 7.4: Struttura in memoria degli eventi

La lista ListaECA La lista listaECA memorizza tutte le regole attive. Ogni record della lista ha la struttura seguente.

```
typedef struct lECA{
char     *name;                /* nome della regola */
lN       *eve;                 /* lista dell'evento della regola */
struct consact  *cons;         /* punt. alla parte condizione */
struct consact  *act;          /* punt. alla parte azione */
struct L_Sigma  *opt;          /* punt. alla parte opzione */
struct lECA     *next;
} lECA;
```

Come si può notare la struttura dati ricalca quella utilizzata nell'odl_trasl, una parte evento, una parte condizione, una parte azione ed una parte opzione. Ancora una volta sia la parte condizione che quella azione fanno riferimento ad una stessa struttura dati che raccoglie tutte le possibili tipologie esprimibili.

```
typedef struct consact{
  char type;                     /* tipologia */
  char *dottedname;               /* nome considerato */
  union
  {
    struct                  /* caso forall */
    {
      char *iter;           /* iteratore */
      struct consact *condition; /* lista condizioni*/
```



```
    } f;
    struct {        /* caso const */
      char *op;     /* espressione */
      char *value;  /* valore */
      char *cast;   /* tipo */
    } c;
    struct {        /* caso type */
      char *type;   /* tipo o classe */
      char *iter;   /* iteratore */
      struct L_sigma   *operation; /* operazione */
    } i;
    struct L_sigma  *oper;  /* caso di esecuzione metodo */
    struct {  /* caso espressione */
      char *second; /* valore di rif. */
      char *op;     /* segno espressione */
      char *value;  /* valore */
    } o;
    struct {  /* caso query */
      char *op; /* espressione */
      char *value; /* valore */
    } q;
  } pp;
struct consact *next;
} consact;
```

Per chiarire meglio si riporta in figura 7.5 un esempio di come viene memorizzata la seguente regola attiva nell'OCDL-designer. Per comodità del lettore si fa riferimento alla regola utilizzata nel paragrafo precedente.

```
ecarule archi1 on delete in Clinical_folder
  if OLD.of.state=''alive''
    do abort
       granularity = instance;
```

7.2.2 Descrizione delle funzioni

- **leggi_schema**
  Funzione che legge il file ocdl contenente lo schema. Essa esegue un ciclo fino a che non incontra il carattere ".", ad ogni passo legge una definizione. Una definizione può essere:
  - un tipo base, in tal caso viene creato un elemento di tipo **bt** e inserito in coda alla lista **listaB**, dopodichè viene chiamata la funzione **leggi_base** che legge la descrizione relativa.

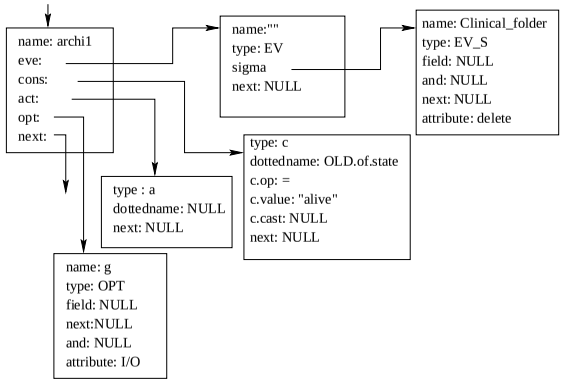


Figura 7.5: Struttura in memoria delle regola attiva archi1

- un tipo valore o di una classe, in tal caso viene creato un elemento di tipo **lN** e inserito in coda alla lista **listaN**, dopodiché viene chiamata la funzione **leggi_descrizione** che legge la descrizione relativa al nome definito.
- una operazione, in tal caso viene creato un elemento di tipo **lN** e inserito in coda alla lista **listaO**, dopodiché viene chiamata la funzione **leggi_operazione** che legge la descrizione relativa alla operazione definita.
- un evento, in tal caso viene creato un elemento di tipo **lN** e inserito in coda alla lista **listaE**, dopodichè viene chiamata la funzione **leggi_evedef** che legge la descrizione relativa all'evento.
- una regola attiva, in tal caso si crea un elemento di tipo **lECA** e inserito in coda alla lista **listaECA**, dopodichè viene chiamata la funzione **leggi_ecarule**

- **leggi_descrizione**
  E' una funzione ricorsiva utilizzata per la lettura della descrizione di un tipo valore o di una classe. Ad ogni chiamata della funzione viene creato un elemento di tipo **L_sigma**, la ricorsione ha termine quando il tipo letto è un tipo base o un nome.

- **leggi_operazione**
  Questa funzione legge la dichiarazione di una operazione. È in grado di leggere il nome della operazione, dopodichè richiama iterativamente la funzione **leggi_descrizione** per ogni parametro. In questo modo si è ottenuta la massima riusabilitá del software.

- **leggi_evedef**
  Questa funzione legge la dichiarazione di un evento. È in grado di leggere tutti i tipi di evento e di memorizzarli nella apposita lista. Per leggere i *Method Event* viene chiamata la funzione **leggi_operazione**. La chiamata a questa funzione si può trovare in due punti nel programma:
  1. direttamente nella lettura dello scherma per la definizione di un *Method Event* o di un *Time Event* ,
  2. dalla funzione per la lettura delle regole quando si è in presenza di un *System Event* .

- **leggi_ecarule**
  Questa funzione legge la dichiarazione di una regola attiva. Essa è in grado di leggere tutti i tipi di regole attive definibili in ODL. Per leggere la parte condizione e la parte azione viene chiamata la funzione **leggi_ca** che effettua chiamate ricorsive quando si trovano più azioni in **AND**.

Le due liste listaE e listaECA non vengono tradotte nella forma canonica perchè non avrebbero nessuna utilità. Vengono però utilizzate attivamente nell'algoritmo di traduzione in UNISQL che verrà trattato nel prossimo paragrafo.

7.3 Run Time di ECA rule

La parte conclusiva del lavoro svolto nella presente tesi è l'implementazione delle regole attive, inserite nello schema, nel DBMS UNISQL. La parte riguardante la traduzione dello schema (classi e rule) dall'ODL alla sintassi di UNISQL è in corsi di sviluppo.
In questi paragrafi si tratta solamente la traduzione delle regole attive. Per una miglior comprensione della metodologia di traduzione si consiglia di leggere l'appendice A dove viene riportata la sintassi di UNISQL riguardanti i *Triggers*.

7.3.1 Traduzione degli eventi

Come già si è fatto notare in UNISQL le regole attive sono denominate "Triggers" e possono essere scatenate dai soli eventi insert, update, delete. Per questo motivo non tutte le regole descritte dalla grammatica dell'ODL esteso (vedi appendice C) possono essere tradotte ed implementate. Vediamo una ad una le singole tipologie di evento.

ODL esteso	UNISQL
Immediate	Before
Delayed	After
Deferred	Deferred

Tabella 7.2: Corrispondenza di Coupling Mode tra ODL esteso e UNISQL

System Event Sono gli unici eventi definiti anche in UNISQL, quindi la traduzione è immediata. Si deve però porre l'attenzione sulla caratteristica **granularity**. Infatti per default questo viene settato come **instance oriented** ma se l'utente specifica **set oriented**, allora si deve anteporre la parola chiave *STATEMENT* all'evento sopra specificato. L'altra proprietà che può essere trattata è il **Coupling Mode** (**EC**) e più precisamente la traduzione viene eseguita come in tabella 7.2. (Per default si considera **EC = Immediate**).

Esempio 21 *Evento segnalato quando viene modificata la classe* Request.

```
...
 on update in Request ...
   granularity = set, EC = Delayed;
```

Viene tradotto

```
CREATE TRIGGER ...
AFTER STATEMENT UPDATE ON Request
...
```

Method Event Gli eventi definiti sull'esecuzione dei metodi non hanno un corrispettivo in UNISQL. Una soluzione possibile, per aggirare il problema, è quella di creare una classe (che non deve essere presente nello schema di partenza) che in questa sede verrà chiamata *Eventclass*. La funzione di questa classe è quella di raccogliere gli eventi che non sono definibili nella sintassi di UNISQL. Gli attributi sono:

nome nome evento.

tipo tipologia di evento

attributo eventuale nome del metodo

classe classe di riferimento

ogg oggetto di riferimento

tempo altre informazioni aggiuntive

Gli oggetti-evento sono creati dai metodi sopra i quali viene definito un evento. Quindi quando nello schema viene trovato un *Method Event* , come ad esempio

```
define event show before Visit.show()
```

nel codice del metodo show() vengono inserite le istruzioni necessarie per creare un oggetto nella classe *Eventclass*. Ovviamente, se si deve segnalare l'evento prima dell'esecuzione del metodo, la parte di codice deve essere aggiunta prima delle funzioni vere e proprie. Se invece si tratta di un evento **after** allora le istruzioni saranno poste in coda. L'oggetto-evento appena creato attiva un Trigger definito sulla *Eventclass* con evento *BEFORE INSERT*. Questo Trigger non fa altro che mandare in esecuzione un metodo "speciale" che gestisce el informazioni delle regole associate all'evento. Questa parte sarà trattata in modo più completo nei paragrafi successivi.
Il metodo in questione si deve anche preoccupare della cancellazione dell'oggetto-evento nella *Eventclass*.

Time Event Nell'implementazione attuale questa tipologia non viene tradotta. Anche per questo caso, però, è possibile utilizzare una traduzione simile a quella precedente, per segnalare un *Time Event* si inserisce un oggetto-evento nella classe *Eventclass* attivando così l'esecuzione del metodo "speciale". La differenza dal caso precedente è che per segnalare un *Time Event* si devono verificare condizioni "temporali" che non dipendono dal DBMS e che possono essere rilevate solo da un processo sempre in esecuzione parallelamente al DBMS. Allora la creazione dell'oggetto-evento deve essere affidata a questo processo.

7.3.2 Traduzione della condizione

La parte condizione in un Trigger UNISQL è introdotta dalla parola chiave *IF* e può essere una qualunque espressione accettabile della grammatica che produce un risultato booleano, quindi *TRUE* o *FALSE*. Anche in questo caso non tutto ciò che è esprimibile con la grammatica dell'ODL esteso è traducibile direttamente in UNISQL.
Nei paragrafi seguenti si fa riferimento solo a regole definite con *System Event* .

True la traduzione si risolve non specificando la parte condizione. UNISQL non trovando la parola chiave *IF* considera il Trigger incondizionato.

Costante Ha come risultato *TRUE* quando l'attributo specificato soddisfa la relazione. La traduzione è diretta, si deve solo cambiare il nome dell'iteratore da **OLD** a **obj** e le doppie virgolette (") in virgolette singole (').

Esempio 22 .

```
... if NEW.nome = "Andrea" ...
```

Viene tradotto

```
...  IF new.nome = 'Andrea'...
```

Forall Può essere tradotto a seconda dell'azione. Questo costrutto non esiste in UNISQL ma se l'azione è un **delete** o un **update** è possibile tradurre il **forall** come una clausola *WHERE* dei comandi.

```
... forall X in Request : X.state="executed"
   do X.delete;
```

Viene tradotto creando un Trigger incondizionato ma specificando una clausola **WHERE** nell'azione.

```
... EXECUTED DELETE FROM REQUEST X
   WHERE X.state='executed';
```

Se invece non è possibile tradurre la condizione in questo modo si deve ricorrere ad un altro stratagemma. Si può creare un Trigger incondizionato che mandi in esecuzione un metodo dedicato all'esecuzione della regola, cioè alla verifica della condizione e all'esecuzione dell'azione. In questo modo, però, si nasconde all'utente il vero comportamento della regola attiva.
Nell'implementazione attuale, il costrutto **forall** non viene tradotto, anche perchè il suo utilizzo è rilevante in combinazione con i *Time Event* che, per il momento, non sono traducibili.

Esecuzione di un metodo Questa tipologia di condizione è traducibile direttamente scrivendo il nome del metodo e, tra parentesi, l'identificatore dell'oggetto su cui eseguire il metodo, seguito dagli eventuali parametri. Come tutti gli altri casi, il risultato deve essere di tipo booleano, quindi se il tipo di ritorno del metodo non è **boolean** è necessario fare un confronto (questo è già previsto dalla grammatica dell'ODL esteso).
In UNISQL il tipo **boolean** non esiste quindi viene tradotto con un tipo integer che può assumere il valore 0 o il valore 1. Siccome nella grammatica dell'ODL esteso per eseguire l'azione si richiede che il risultato della condizione sia *True*, nel tradurre i metodi di tipo **boolean** si farà sempre il confronto con 1.

Esempio 23 .

```
... if NEW.non_esente (NEW.name) do ...
```


Viene tradotto

```
...  IF non_esente(NEW[2], name[3]) = 1 ...
```

Per quanto riguarda la parte condizione e la parte azione definite assieme a *Method Event* , si è già detto che la loro gestione viene affidata al metodo "speciale" che viene attivato ad ogni inserimento dell'oggetto-evento nella classe *Eventclass*.

Oltre alla classe riservata agli eventi si crea anche una classe riservata alle regole chiamata *ECAclass* che viene utilizzata per memorizzare la parte condizione e la parte azione delle regole attive che utilizzano gli oggetti-evento per essere attivate. La classe *ECAclass* viene popolata da oggetti-regola al momento della traduzione delle regole attive. Infatti, quando si incontra un *Method Event* si inserisce nella classe *ECAclass* un oggetto che contiene tutte le informazioni sulla regola attiva utilizzate dal metodo "speciale" che esegue effettivamente la regola. In questo modo non si nasconde all'utente il vero comportamento del database attivo, perchè le informazioni sono reperibili dagli oggetti-regola, anche se la sua gestione è affidata a un metodo. Purtroppo è necessario utilizzare questa metodologia di traduzione con tutte le regole attive che fanno riferimento a *Method Event* e quindi anche quelle che possiedono condizione e azione che sarebbero traducibili direttamente.

7.3.3 Traduzione dell'azione

La parte azione è quella dove è possibile il maggior numero di traduzioni dirette. Di seguito vengono trattate una ad una le tipologie definibili dalla grammatica dell'ODL esteso.

Abort si traduce in due modi:

1. *REJECT* impedisce che l'eventuale modifica, inserimento o cancellazione che ha causato l'attivazione del Trigger sia portata a termine
2. *INVALIDATE TRANSACTION* impedisce di eseguire il **commit** della transazione e quindi tutte le eventuali modifiche vengono annullate.

Costante Si traduce direttamente con il comando *UPDATE OBJECT* se la condizione non è il costrutto **forall**, in caso contrario, si utilizza il solo *UPDATE*. Vengono usati gli stessi accorgimenti discussi nella traduzione della parte condizione. Si veda l'esempio per ulteriori chiarimenti.

Esempio 24 .

[2] Identificatore di oggetto
[3] Parametro del metodo

```
... if true do O.state="executed";
```

Viene tradotto

```
... EXECUTE UPDATE OBJECT OBJ
    SET state='executed';
```

Oppure si consideri

```
... if forall X in Request : X.state="executed"
  do X.delete;
```

Si traduce

```
... EXECUTED DELETE FROM REQUEST X
    WHERE X.state='executed';
```

Esecuzione di un metodo Si traduce direttamente con l'istruzione *CALL* che permette la chiamata ai metodi. Dopo la parola chiave deve essere inserito il nome del metodo, i parametri e, in ultimo, l'identificatore dell'oggetto su cui il metodo deve essere eseguito.

Esempio 25 .

```
... do NEW.print(1);
```

Viene tradotto

```
... EXECUTE CALL print(1) INTO new;
```

Espressione Si traduce direttamente facendo un **update** sull'oggetto in questione. L'istruzione *UPDATE OBJECT* necessita dell'identificatore dell'oggetto su cui si deve effettuare la modifica seguita dalla parola chiave *SET*. Se questa tipologia è usata in concomitanza con il costrutto **forall** allora si deve usare l'istruzione *UPDATE* seguita dalla clausola *WHERE*.

Esempio 26 .

```
... do O.ticket = O.ticket*1,1;
```

Viene tradotto

```
... EXECUTED UPDATE OBJECT obj
    SET ticket = ticket*1,1;
```


Delete Si traduce direttamente con il comando *DELETE* di UNISQL, specificando nella clausola *WHERE* quale oggetto si vuole cancellare.

Tutte le informazioni relative allo schema e alle regole attive tradotte in UNISQL sono raccolte nel file *nomeschema.c* e nel file *nomeschema_met.ec* (che mantiene il corpo dei metodi).
Il primo deve essere compilato e l'eseguibile che viene prodotto deve essere eseguito con il nome del database di UNISQL come parametro (es: clinic.exe provadb). Il file dei metodi deve essere precompilato e poi compilato, ma non si deve produrre l'eseguibile. Infatti UNISQL accetta solo la forma oggetto.
Si vuole sottolineare che l'UNISQL permette di definire una sola azione per Trigger, quindi le regole attive che nella parte azione utilizzano l'operatore **and** vengono tradotte con tanti Trigger quante sono le parti in **and**. Un altro limite del DBMS UNISQL è il gran numero di parole chiave come (name, by, of, ecc.) che non possono essere utilizzate nello schema o nelle regole attive del database che l'utente vuole creare. Per maggior chiarezza suoi passi da seguire per la creazione di un database con regole attive si veda l'esempio di sessione di lavoro nel capitolo 9

Capitolo 8

Terminazione e Confluenza di regole attive

8.1 Analisi del comportamento delle regole attive

Come si è mostrato nei capitoli precedenti le regole attive possono essere usate per diversi scopi: mantenimento dell'integrità, workflow management, replicazioni e così via. Per esempio se le regole ECA sono utilizzate per controllare l'integrità del sistema nel momento in cui l'utente inserisce dei dati che violano l'integrità, le regole reagiscono dinamicamente facendo il rollback della transazione o correggendo l'errore. In generale le regole attive impongono un "regolamento dinamico" al database a cui sono applicate.
In tutti i casi, però, interagiscono tra loro ed è propio questo che può generare dei problemi. E' molto difficile prevedere il comportamento di un set di regole perchè non si è in grado di stabilire esattamente quando e in quale ordine vengano attivate ed eseguite le ECA rule. Su questa materia sono stati fatti diversi studi ed in particolare sono state riconosciute due proprietà fondamentali che ogni set di regole dovrebbe avere: *terminazione* e *confluenza*.
La proprietà di terminazione si ottiene nel momento in cui il processo di esecuzione delle regole ha un termine, cioè le regole si attivano a vicenda un numero finito di volte. La proprietà di confluenza si ottiene quando lo stato finale del database non dipende dall'ordine di esecuzione delle regole. Diversi approcci sono stati studiati e proposti per garantire queste due proprità, dalla semplice analisi sintattica [**?**] alla complessa analisi semantica [**?**, **?**].
Di seguito si presentano alcuni di questi approcci e infine si illustra quale algoritmo è stato implementato in ODB-Tools.

8.2 Terminazione

8.2.1 Analisi semplice

Da [**?**] viene riportata la parte riguardante la terminazione. Il testo originale fa riferimento al DBMS Starburst ma l'algoritmo descritto può essere considerato anche in termini generali facendo riferimento alla grammatica dell'ODL esteso. Sia R=$\{r_1, r_2, ..., r_n\}$ un arbitrario set di regole attive che devono essere analizzate. L'analisi viene svolta su un set di regole fissato, se il set viene modificato anche l'analisi deve essere rifatta. Sia O un insieme di operazioni di modifica del database. In O vanno inserite tutte le operazioni di insert, delete, update ed esecuzione di un metodo[1].
Segue la descrizione di alcune funzioni che vengono eseguite come analisi preliminare:

- *Triggered-By* prende una regola r e ritorna il set di operazioni in O che attivano r.
- *Performs* prende una regola r e ritorna il set di operazioni in O che vengono eseguite dall'azione della regola r.
- *Triggers* prende una regole r e ritorna tutte le regole r' che vengono attivate dall' esecuzione delle azioni di r. Triggers(r) = { r' $\in R$ | Performs(r) $\cap$ Triggered-By(r)$\neq \emptyset$ }
- *Abort* prende una regola r e indica se produce un abort della transazione.

Grazie a queste funzioni si può a costruire il *Triggering Graph* per il set di regole R, chiamato TG_R. Il TG_R ha un nodo per ogni regola r $\in$ R tale che *Abort(r)=false* e un arco che collega due nodi r_i a r_j se e solo se r_j $\in$ *Triggers*(r_i). Cioè se r_i compie un'azione che attiva r_j, allora il TG_R avrà un arco che va dal nodo r_i al nodo r_j. Il grafo che si costruisce, seguendo le regole appena citate, serve per scoprire tutte le relazioni che legano le regole per verificare se la proprietà di terminazione è rispettata oppure no. Basta esaminare se nel TG_R sono presenti dei cicli, infatti se vengono trovati dei cicli si può affermare che l'esecuzione di R *può* non terminare; le motivazioni del *può* saranno chiarite più avanti. Poichè cerchiamo dei cicli infiniti possiamo escludere le regole che causano un abort, anche se hanno *Triggers*(r_i)$\neq \emptyset$[2], perchè tutti gli effetti della transazione vengono annullati.
Cerchiamo di capire perchè un ciclo in TG_R comporta il fatto che R non gode della proprietà di terminazione. Supponendo che R sia composto da un numero finito di regole, se l'esecuzione di R non termina significa che almeno una regola r è eseguita un numero infinito di volte. La regola r, per essere eseguita, deve essere attivata

[1] Questo ovviamente vale solo per DBMS ad oggetti
[2] Cioè attivano altre regole.

da un'altra regola cioè $r \in$ *Triggers(r_i)*, e quindi un arco da r_i a r; r_i, però, può attivare r un numero finito di volte a meno che anch'essa non sia chiamata una infinità di volte. Quindi deve esistere una r_j tale che $r_i \in$ *Triggers(r_j)*, e quindi esisterà un arco da r_j a r_i in TG_R. Procedendo con il ragionamento si deduce che essendo R composto da un numero finito di regole si deve avere un ciclo in TG_R perchè r sia eseguita un numero infinito di volte.
Dunque il problema della Terminazione si riduce nell'analizzare se esistono cicli in TG_R. Vediamo un esempio di algoritmo adibito alla creazione e all'analisi del grafo.

- Riduco R $\rightarrow$ R': elimino r $\in$ R se solo se *Abort(r)=true*
- Eseguo *Triggered-By(r)* $\forall$ r $\in$ R'
- Eseguo *Performs(r)* $\forall$ r $\in$ R'
- Eseguo *Triggers(r)* $\forall$ r $\in$ R'
- Costruisco TG_R
- Esamino TG_R per trovare cicli

Come si può notare nel formare il TG_R non si tiene conto in alcun modo della parte condizione, ci si pone infatti nel caso in cui questa sia sempre verificata e che quindi la parte azione sia sempre eseguita. E' facile, dunque, che l'algoritmo segnali la presenza di un ciclo infinito quando invece ciò non si verifica. Per esempio se una regola compresa nel ciclo non fa altro che un *delete* su una classe e nessun'altra regola del ciclo effettua un *insert* sulla stessa classe, allora questo ciclo è destinato a terminare quando non ci saranno più oggetti. Oppure, ancora, l'azione di una regola non fa altro che incrementare un contatore che prima o poi renderà falsa la condizione della regola. Quindi l'interazione dell'utente è necessaria per una corretta interpretazione dei risultati di questo algoritmo. È comunque possibile costruire una routine che rileva queste casistiche automaticamente, ed evita di segnalare un ciclo infinito. In fase di analisi dello schema non è possibile verificare la veridicità o meno di una condizione è solo possibile fare delle supposizioni.
È possibile migliorare l'analisi inserendo anche lo studio delle precedenze tra le regole definite dall'utente con le opzioni precedes e follows, però si deve tenere presente che non tutti i database implementano la possibilità di dare un ordine di esecuzione alle regole.
Altro aspetto che non può essere tenuto in conto dall'algoritmo di terminazione è l'effetto collaterale dell'esecuzione di un metodo. Infatti all'interno di una operazione possono essere fatte modifiche al database causando l'attivazione di altre regole. Tutto ciò non può essere rivelato dall'algoritmo perchè non c'è modo si sapere quali operazioni vengono eseguite all'interno dei metodi. Ecco spiegato il perchè il risultato dell'algoritmo sono i *possibili* cicli infiniti.
Rifacendosi all'esempio della Clinica riportato nel paragrafo 6.1.1, si può notare

come a volte si creino i presupposti per cicli infiniti senza accorgersene. Una regola di questo tipo produce un ciclo infinito:

```
ecarule ciclo on insert in Visit if true
     do NEW in Request;
```

Infatti la regola ad ogni inserimento nella classe Visit produce un inserimento nella classe Request, che è figlia della stessa classe Visit. Però il paradigma ad oggetti impone che le regole siano ereditate dalle classi figlie, quindi ad ogni insert nella classe Visit si produce un insert nella classe Request che fa attivare nuovamente la regola. Il sistema entra così in un ciclo infinito.

8.2.2 Analisi Complessa

In questo paragrafo viene riportato uno studio più approfondito sulla terminazione [**?**] che oltre ad osservare quale evento può attivare una regola si preoccupa di scoprire se le azioni delle regole possono rendere vera la condizione di altre regole. Si definiscono dunque due grafi: il *Triggering Graph* (TG) e l'*Activation Graph* (AG).
Supponiamo di avere un set qualsiasi, R, di regole attive, il TG è un grafo composto da nodi e archi (V,E), tali che ogni nodo $v_i \in$ V corrisponde a una regola $r_i \in$ R, ogni arco (v_j,v_k) $\in$ E significa che l'azione di r_j genera un evento che attiva la regola r_k. Facendo sempre riferimento a R, l'AG è un grafico composto da nodi e archi (V,E). Anche in questo caso i nodi identificano le regole, mentre un arco (v_j;v_k) con j$\neq$k significa che le regola r_j può cambiare il valore della condizione di r_k da falso a vero, cioè la condizione di r_k può risultare vera dopo l'esecuzione di r_j.
Per scoprire se il set R gode della proprietà di terminazione ho bisogno di tutti e due i grafi. Però, mentre il primo (TG) è derivabile sintatticamente dalla definizione delle regole, il secondo (AG) non si determina così facilmente. A volte è necessario inserire un arco in una visione pessimistica perchè risulta difficile o addirittura impossibile conoscere il vero valore della condizione.
Si concentra l'attenzione sulle regole senza priorità presentando alcune osservazioni concernenti la loro terminazione.
Sia R un set finito, e arbitrario di regole attive. Durante la fase di esecuzione ogni regola r $\in$ R è eseguita più di una volta solo se r ha almeno un arco entrante sia in TG che in AG. Infatti, perchè una regola sia eseguita deve essere innanzi tutto attivata da un evento e poi la sua condizione deve risultare vera. Dopo la sua esecuzione la regola viene disattivata, perchè possa essere eseguita nuovamente un'altra regola r' $\in$ R deve generare un evento che la riattivi. Quindi deve esistere un arco in TG (r',r). Dopo la prima esecuzione la condizione di r può essere diventata falsa, allora perchè sia veramente eseguita si deve avere una regola r" $\in$ R che faccia diventare la condizione da falsa a vera, cioè deve esistere un arco (r",r) in AG. In questo modo la regola può essere eseguita più di una volta.

Da ciò è possibile dedurre un algoritmo di riduzione, cioè un processo che elimina delle regole da R che non potranno mai appartenere ad un ciclo.

```
Ripeti fino a che sono possibili eliminazioni
 if (nessun arco(r',r) in TG) or (nessun arco(r",r) in AG)
    Elimina r da R, da TG, da AG.
```

L'insieme che si ottiene da questa riduzione viene detto *Irriducibile* o IR. In pratica ad ogni giro l'algoritmo elimina le regole che sono eseguite un numero finito di volte. Infatti, al primo giro vengono eliminate le regole che possono essere attivate solo da una transazione dell'utente e che quindi saranno eseguite solo una volta. Dal secondo giro, l'algoritmo elimina le regole che sono attivate da quelle già eliminate al giro precedente e che quindi sono destinate ad essere attivate un numero finito di volte.
Dunque, se l'algoritmo di riduzione produce un insieme vuoto, IR=$\emptyset$, allora R gode della proprietà di terminazione. Invece se IR$\neq \emptyset$, allora le regole al suo interno possono partecipare ad un ciclo infinito. Se tra le regole appartenenti al set IR esiste un ciclo sia in TG che in AG, allora r non gode della proprietà di terminazione.

8.3 Confluenza

Quando ci si chiede se un set di regole R=$\{r_1, r_2, ..., r_n\}$ è confluente, si vuole determinare se lo stato finale del database, al termine del processo di esecuzione, può dipendere dall'ordine seguito nel prendere in considerazione le regole.
Un semplice esempio può essere:

```
ecarule r1 on insert in Request if true
     do NEW.ticket= integer cl_ticket(NEW.by.number_card);
ecarule r2 on insert in Request
      if NEW.n_esente(NEW.by.number_card)
         do NEW.ticket = NEW.ticket * 1.1;
```

La prima regola calcola il costo del ticket per la visita richiesta, mentre la seconda aumenta del 10% il ticket in caso che il richiedente non sia esente. Quando viene effettuato un insert nella classe Request, le due regole vengono attivate. Se viene eseguita la prima e poi la seconda il risultato è giusto, ma se l'ordine è opposto allora l'aumento del 10% viene effettuato su un valore che non è corretto, inoltre viene comunque sostituito con il valore calcolato dal metodo. Per risolvere un problema di questo tipo basta inserire un ordine di precedenza tra le regole.
Un esempio più subdolo e difficile da scoprire è il seguente. Supponiamo di dover gestire un negozio. Si vuole premiare i commessi a seconda del numero dei pezzi venduti secondo le due regole seguenti:

1. Se un commesso effettua una vendita superiore ai 20 pezzi riceve un aumento di stipendio di 10.
2. Se un commesso effettua una vendita superiore ai 50 pezzi riceve, oltre all'aumento di stipendio di 10, un punto di livello. Quando raggiunge il livello 5 avrà uno aumento del 20% dello stipendio.

Nella sintassi delle ECA rule:

```
ecarule bonus on insert in Vendite if NEW.qty > 20
    do NEW.by.salary = NEW.by.salary + 10;
ecarule rank on insert in Vendite if NEW.qty > 50
    do NEW.by.rank = NEW.by.rank + 1;
ecarule bonus_rank on update in Employee if NEW.rank = 5
    do NEW.salary = NEW.salary * 1.2;
```

Per evitare errori si deve imporre la precedenza di *bonus_rank* sulle altre due regole, in modo tale che in presenza di inserimenti multipli su Vendite venga per primo fatto l'incremento del 20% e poi gli eventuali aumenti di stipendio e di livello. Quindi all'ultima definizione andrà aggiunto:

```
precedes(bonus, rank);
```

Però non si è risolto completamente il problema infatti ad una attenta osservazione si nota che se un commesso, di stipendio 100 e livello 4, effettua una vendita superiore a 50 vengono attivate sia bonus che rank. Allora se viene eseguita prima *bonus* si avrà uno stipendio di 120 e poi, dopo l'esecuzione di *rank* e, di conseguenza, anche di *bonus_rank*, si avrà uno stipendio di 144. Se invece viene eseguito prima *rank* si avrà uno stipendio di 120 poichè viene attivata anche *bonus_rank* che ha precedenza su tutte. Alla fine dell'esecuzione il salario sarà di 140, diverso da quello precedente. Questo per dimostrare che da due regole che apparentemente non hanno nulla in comune si ottengono due stati finali del database diversi.

8.4 Realizzazione dell'algoritmo di Terminazione per ODB-Tools

L'algoritmo di terminazione che si è voluto implementare in ODB-Tools ricalca quello spiegato nel paragrafo 8.2.1. La grammatica disponibile all'utente per costruire il set di regole è sufficiente per generare dei cicli che sfuggono ad una prima analisi. Quindi può risultare utile avere uno strumento che segnali eventuali cicli. È stato però inserito anche l'algoritmo di riduzione che riduce il set in esame togliendo le regole che non possono far parte di nessun ciclo perchè sono attivate solo dalle transazioni dell'utente e quindi un numero finito di volte.

- Riduco R → R': elimino r ∈ R se solo se *Abort(r)=true*
- Eseguo *Triggered-By(r)* ∀ r ∈ R'
- Eseguo *Performs(r)* ∀ r ∈ R'
- Eseguo l'Algoritmo di Riduzione
  - Ripeti fino a che ci sono regole eliminate
  - Eseguo *Triggers(r)* ∀ r ∈ R'
  - Riduco R' → R": elimino r_j ∈ R' se solo se $r_j \notin$ *Triggers*(r_i) per i=1,...,n.
  - R' = R".
- Costruisco TG_R
- Esamino TG_R per trovare cicli

8.4.1 Le strutture dati

L'algoritmo memorizza tutti i suoi dati in una unica lista:

```
struct ecatr_list
 {
  char flag;
  char *name;
  struct ecatr_act *eve;
  struct ecatr_act *act;
  struct ecatr_trig *trig;
  struct ecatr_list *next;
 };
```

Il significato dei vari campi è:

- FLAG: serve per marcare la regola quando viene esaminata dall'algoritmo di terminazione. In questo modo se esamino una regola marcata sono in presenza di un ciclo.
- name: nome della regola
- EVE: puntatore alla lista che mantiene gli eventi che causano l'attivazione della regola.
- ACT: puntatore alla lista che mantiene le azioni fatte dalla regola
- TRIG: puntatore alla lista che mantiene i nomi delle regole che vengono attivate dalle azioni.

- NEXT: puntatore alla regola seguente.

Gli elementi delle lista *ecatr_act* hanno la seguente struttura:

```
struct ecatr_act
 {
  char type;
  char *cname;
  char *oname;
  struct ecatr_act *next;
 };
```

I campi hanno i seguenti significati:

- TYPE: il tipo di evento e azione memorizzato. Può assumere i valori:

  m : esecuzione metodo;

  t : Time event; solo per gli eventi

  i : insert;

  u : update;

  d : delete;

  a : abort; solo per le azioni;
- CNAME: mantiene il nome della classe
- ONAME: mantiene il nome del metodo quando necessario.

In ultimo la lista *ecatr_trig* che mantiene la lista dei nomi delle regole che vengono attivate dalle azioni della regola stessa.

```
struct ecatr_trig
 {
  char *name;
  struct ecat_trig *next;
 }
```

In figura 8.1 si mostra come viene memorizzata la regola attive **start.** Dopo la prima esecuzione della funzione *Triggers* viene riempita anche la lista *trig*.

```
ecarule start on morning if forall X in Request :
        X.date_fixed = string today()
              do X in To_day
```

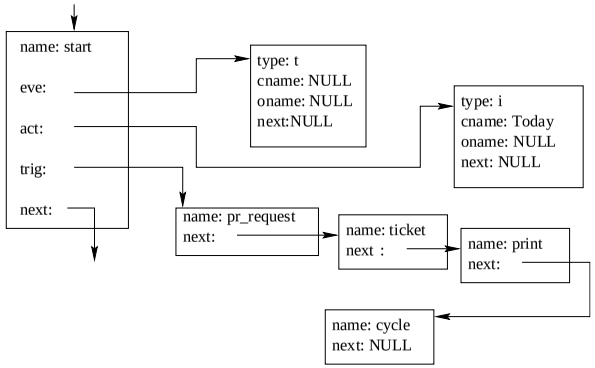


Figura 8.1: Esempio di memorizzazione di una regola per l'algoritmo di terminazione

8.4.2 Descrizione delle funzioni

Si riportano le funzioni che implementano l'algoritmo di terminazione e che fanno parte del programma odl_trasl:

eca_tr Funzione principale che chiama tutte le altre e controlla che non ci siano errori. Dopo aver riempito la lista principale chiama ciclicamente le due funzioni che compongono l'algoritmo di riduzione (**find_triggered, riduzione**). Infine, nuovamente con un ciclo esamina l'insieme di regole rimasto per individuare un eventuale ciclo.

eca_view Esamina una ad una le regole attive definite nello schema e determina da quali azioni possono essere attivate e quali azioni effettuano. In questa funzione si tiene conto anche dell'ereditarietà delle regole.

find_triggered Riempie la lista *trig* di ogni regola. Cioè ogni regola mantiene in memoria quali altre regole attiva eseguendo le sua azioni. Praticamente in questa funzione viene costruito il *Triggering Graph*. Ogni volta che trovo una regola che viene attivata da una azione la segno con un "*" nel campo flag delle lista ecatr_list.

riduzione E' l'implementazione dell'algoritmo di riduzione. Cerca tutte le regole che non vengono attivate dalle altre regole. In pratica effettua un esame su tutte le regole del *Triggerin Graph* cercando quelle che non hanno un ramo in ingresso. Per fare questo si utilizza il campo flag della ecatr_list, se viene

trovato un "*" la regola deve rimanere, se viene trovato " " allora si toglie la regola dalla lista.

find_cycle Questa funzione chiamandosi ricorsivamente individua gli eventuali cicli infiniti. Non fa altro che verificare se le regole nel campo *trig* della regola in esame, non sono già state prese in considerazione (ancora una volta si utilizza il campo flag). Si parla di eventuali cicli perchè non si tiene in conto della parte condizione della regola. Infatti non è possibile verificare se la condizione è vera o falsa, quindi si assume che sia sempre vera. Se si individua un ciclo lo si segnala all'utente che deve verificare se effettivamente è infinito oppure no.

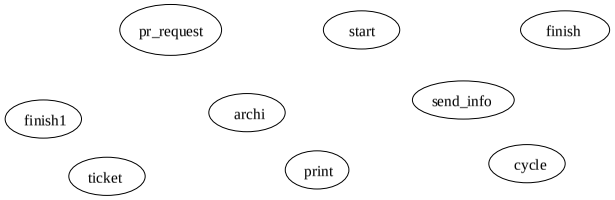


Figura 8.2: Nodi del grafo TG_R

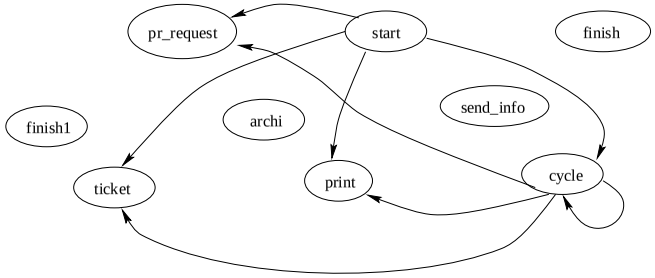


Figura 8.3: Nodi e archi del grafo TG_R

8.5 Esempio di funzionamento per lo schema CLINIC

Facendo riferimento al set di regole dello schema CLINIC introdotto nel paragrafo 6.1.6 si aggiunge la regola *cycle* per mostrare come vengono rilevati eventuali cicli.

```
ecarule cycle on insert in Visit if true
      do NEW in Request;
```

Nella figura 8.2 si mostrano i nodi creati con le regole dello schema. Si nota come le regole *show_eca* e *archi1* non vengano creati perchè l'azione di queste regole prevede un **abort**. Quindi non potranno mai partecipare ad un ciclo e vengono eliminate a priori.
Dopo l'esecuzione della funzione *Triggers* si definisce il rapporto fra le varie regole. Vengono creati gli archi tra i nodi del grafo, vedi figura 8.3. A questo punto viene mandato in esecuzione l'algoritmo di riduzione che elimina tutti i nodi che non hanno archi in ingresso (figura 8.4). Siccome alcuni nodi sono stati eliminati si

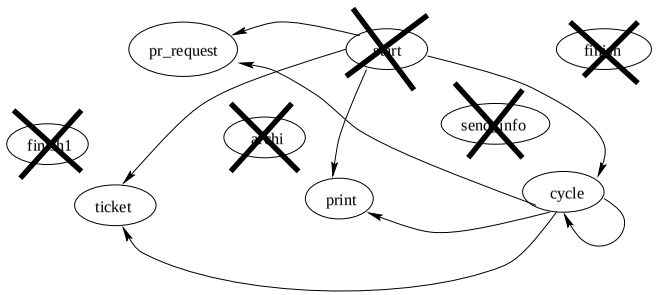


Figura 8.4: Nodi e archi del grafo TG_R dopo l'esecuzione dell'algoritmo di riduzione

rimanda in esecuzione la funzione *Triggers* per eliminare eventuali relazioni ormai inutili. Il risultato di questo secondo ciclo viene mostrato in figura 8.5. L'algoritmo di terminazione non riesce più ad eliminare nulla perchè tutti i nodi hanno almeno un arco in ingresso. Allora viene mandata in esecuzione la funzione dedicata alla ricerca dei cicli. L'unico ciclo trovato è quello sulla regola *cycle*. Quindi il uscita l'odLtrasl segnalerà la presenza du una possibile esecuzione infinita sulla regola *cycle*.
L'esempio riportato è stato creato appositamente per mostrare il funzionamento dell'algoritmo. Un attento osservatore sarebbe riuscito a trovare l'errore subito, questo solo perchè ancora la grammatica a disposizione dell'utente non è vastissima e quindi è facile tenere sotto controllo il set di regole attive. Si pensi, però, ad un database che necessiti di un centinaio o più di regole attive, un algoritmo di rilevazione di eventuali cicli infiniti diventa indispensabile perchè il rulebase diventa ingestibile.

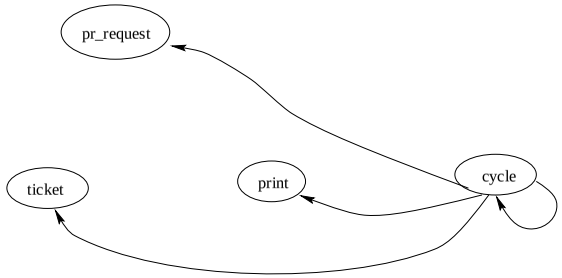


Figura 8.5: Nodi e archi del grafo TG_R che compongono l'insieme IR (irriducibile)

Capitolo 9

Conclusioni e sviluppi futuri

9.1 Esempio di sessione di lavoro

La prima operazione da eseguire è a carico dell'utente il quale genera il file contenente lo schema del database scritto nella sintassi dell'ODL esteso (si veda appendice C) chiamandolo *nomeschema.odl*, nell'esempio di questo paragrafo il file in questione sarà composto dallo schema introdotto nel capitolo 6 e dalle regole introdotte nella sezione 6.1.6. Per mandare in esecuzione il Traduttore occorre digitare il comando *odl_trasl* seguito dal file contenente lo schema scritto dall'utente. Se quest'ultimo viene omesso, il programma si aspetta che l'utente inserisca lo schema direttamente da tastiera. Subito dopo aver lanciato il traduttore vengono visualizzati una serie di messaggi che indicano lo stato di esecuzione:

Esempio 27 *Traduzione dello schema clinic.odl*

```
>odl_trasl clinic
Processing schema
Schema OK
Processing event
Event  OK
Processing eca rules
Ecarule  OK
Processing Rule
No Rule Definitions
Searching in Ecarules for possible infinite loop
Searching End
>
```

Il programma è terminato correttamente e non sono stati rilevati errori durante il processo di verifica della semantica e durante la traduzione nella sintassi dell'OCDL.

Se all'interno dello schema sono presenti regole con parte condizione contenente una query, questa deve essere ottimizzata, se possibile. Allora il traduttore chiamerà l'OCDL-Designer e l'ODBQ-Optimizer in sequenza e l'output sarà:

Esempio 28 *Traduzione dello schema nel caso in cui siano presenti query nelle regole attive.*

```
>odl_trasl clinic
Processing schema
Schema OK
Processing event
Event  OK
Processing eca rules
Ecarule  OK
Processing Rule
No Rule Definitions
*************** OCDL-DESIGNER ***************
Source Schema Acquisition: OK
Canonical Form: OK
Subsumption: OK
Writing Output Files (ese/clin.fc,ese/clin.vf): OK
*************************************************
Query optimized
Searching in Ecarules for possible infinite loop
Searching End
>
```

L'OCDL-Designer viene chiamato con l'opzione (-n) per generare solo il file contenente lo schema in forma canonica e non tutti gli altri file. L'ODBQ-Optimizer viene chiamato con l'opzione (-n) per avere la query ottimizzata scritta su un file in semplice testo, altrimenti l'output sarebbe solo su video.
I risultati dell'elaborazione vengono scritti in due file:

- `nomeschema.sc` contiene la traduzione dello schema del database in OCDL, delle operazioni e, infine, degli eventi e delle regole in una forma meno leggibile ma più maneggevole.
- `nomeschema.vf` contiene i dati necessari all'applet **scvisual** per visualizzare graficamente lo schema.

Nel caso vengano rilevati errori, questi sono segnalati con un messaggio che indica il tipo di errore riscontrato.
In ultimo se l'algoritmo di terminazione trova un possibile ciclo infinito lo segnala mostrando le regole coinvolte

Esempio 29 *Traduzione di uno schema contenente errori.*

```
>odl_trasl clinic
Processing schema
Schema OK
Processing event
Event  OK
Processing eca rules
Not found var [stat] in interface Request
Not found type of [X.stat]
Errors in forall condition
Errors in ECARULE condition finish1
Processing Rule
No Rule Definitions
>
```

Traduzione di una schema contenete un possibile ciclo infinito

```
...
Searching in Ecarules for possible infinite loop
cycle cycle
Possible Loop Found
```

Dopo aver eseguito la traduzione, per completare i controlli sullo schema e per eseguire l'implementazione si deve mandare in esecuzione l'OCDL-Designer. Si utilizza il comando *ocdl-designer* senza opzioni in modo tale da avere come output tutti i file. Prima di eseguire il comando, si deve inizializzare la variabile di ambiente UNISQL_METHOD che deve contenere il path dove il programma andrà ad inserire il file contenete il corpo dei metodi.
Immediatamente il processo chiede all'utente di inserire il nome del file su cui eseguire l'elaborazione, questo deve essere *nomeschema.sc* prodotto dal traduttore.

Esempio 30 *Compilazione dello schema con regole.*

```
>setenv  UNISQL_METHOD=~/body
>ocdl-designer
*************** OCDL-DESIGNER ***************
Schema file name : clinic
Source Schema Acquisition: OK
Canonical Form: OK
Subsumption: OK
Writing Output Files (ese/clinic.fc,ese/clinic.vf): OK
Writing Output File (ese/clin.c):
Eca rule [send_info] not translated
Eca rule [ticket] not translated
```



```
Eca rule [finish1] not translated
Eca rule [finish] not translated
Eca rule [start] not translated
Eca rule [show_eca] not translated
OK
*********************************************
```

Gli eventuali errori vengono segnalati e spiegati. Inoltre viene comunicato quali regole attive non sono state tradotte nella sintassi di UNISQL e che quindi non verranno implementate.
Se non vengono rilevate incoerenze i risultati dell'elaborazione vengono scritti nei seguenti file:

- `nomeschema.fc` contiene per ciascun nome dello schema, compresi i nuovi nomi creati durante il procedimento di generazione della forma canonica, la descrizione originale, la descrizione in forma canonica e la lista dei nomi delle generalizzazioni.
- `nomeschema.sb` contiene tutte le relazioni di sussunzione calcolate e le relazioni *isa*.
- `nomeschema.c` contiene tutte le istruzioni per la generazione dello schema in UNISQL. È un vero programma scritto in linguaggio C.
- `nomeschema_met.ec` contiene tutti i body dei metodi definiti nello schema. Nell'implementazione attuale, nello schema viene inserita sola la signature dei metodi, di conseguenza nel file si avranno tutti i metodi vuoti.

Per poter utilizzare effettivamente il file generati come output è necessario compilarli ed eseguirli. Per compilare il primo (*nomeschema.c*) si devono seguire i seguenti passi[1]:

Esempio 31 *Creazione dell'eseguibile che implementerà lo schema in UNISQL.*

```
>gcc -I$UNISQLX/include -c clinic.c
>gcc $1.o -o clinic.exe -lsqlxone -lsqlxutil -lm
```

Dopodichè si deve generare fisicamente il database seguendo le procedure proprie di UNISQL.

Esempio 32 *Creazione del database di prova e implementazione dello schema Clinic.*

```
>createdb prova
>clinic.exe prova
```

[1] nomeschema = clinic

Il file (*nomeschema_met.ec*) deve essere prima precompilato e poi compilato per produrre la sola forma oggetto che viene poi direttamente utilizzata dal DBMS UNISQL.

Esempio 33 *Compilazione del file contenete il corpo dei metodi.*

```
>esqlx clinic_met.ec
>gcc clinic_met.c -c -I$UNISQLX/include -L$UNISQLX/lib
```

A questo punto il lavoro è concluso e si può lavorare sul database creato inserendo le istanze nelle classi generate.

9.2 Conclusioni e sviluppi futuri

Nella presente tesi è stato studiato lo stato dell'arte nel campo delle basi di dati attive con particolare attenzione alle diverse implementazioni.Si è presentato il progetto di estensione della grammatica dell'ODL (*Object Definition Language*) dallo standard per ODBMS ODMG-93 in modo tale da supportare la definizione di regole attive direttamente nello schema del database. Nel progettare tale estensione, si è cercato di mantenere la sintassi più semplice e più intuitiva possibile.
Si è realizzato un ambiente al top di un OODBMS commerciale
Rapportando il risultato finale rispetto ai canoni definiti per gli ADBMS nel paragrafo 2.1 si può affermare che l'ambiente sviluppato nella presente tesi gode delle seguenti proprietà:

- L'ambiente sviluppato è conforme a ODMG-93 e al top di un OODBMS tradizionale estendendone quindi le funzionalità in senso attivo.
- Tramite la l'ODL esteso è possibile definire eventi, condizioni, azioni e controllare il comportamento delle regole tramite alcune proprietà.
- Il modello di esecuzione è quello del DBMS UNISQL.
- Il rilevamento degli eventi, la verifica delle condizioni e l'esecuzione delle azioni sono, quando possibile, lasciate al DBMS UNISQL. Nei casi in cui il DBMS di appoggio non può intervenire la gestione del comportamento attivo viene lasciata a metodi dedicati.
- Se l'utente ignora le funzionalità attive, il sistema tratta lo schema come database passivo.
- Il problema della gestione dell'evoluzione del set di regole non viene trattato dalla presente tesi perchè non riguarda la definizione dello schema.
- Analisi della proprietà di terminazione del *rulebase*

Per quanto riguarda l'implementazione delle regole attive sul sistema DBMS UNISQL si aggiunge che nella presente tesi si è esteso un modulo software ancora in via di sviluppo e quindi incompleto in alcune sue parti. Il risultato ottenuto può dunque essere ampiamente migliorato.

Dal punto di vista della grammatica dell'ODL esteso si possono aggiungere alcune funzionalità importanti come la possibilità di definire eventi e condizioni composte da operatori del tipo OR, AND, NOT, ecc. Questo comporterebbe anche l'ampliamento della parte adibita alla gestione delle regole attive che dovrebbe implementare i *net-effects*.

L'algoritmo di terminazione può essere potenziato aggiungendo il controllo della precedenza tra le regole attive. Inoltre potrebbe essere aggiunto uno studio sulla parte condizione per determinare effettivamente se un ciclo può o no essere infinito.

Il modulo software adibito alla traduzione delle regole attive nella grammatica di UNISQL può essere migliorato ed ampliato in tutte le sue parti.

Lo stesso modulo può inoltre essere reso indipendente dall'ocdl-designer per costituire un modulo a sè stante in modo da permettere lo sviluppo di altri traduttori per altri DBMS commerciali. Così facendo si permette all'utente di scegliere su quale DBMS implementare lo schema.

Appendice A

Grammatica di definizione dei Triggers in UNISQL

Un Trigger può essere creato per invocare un'azione in risposta a una specifica attività sul database. Quando questo evento viene rilevato il Trigger definito viene scatenato. La grammatica per la definizione dei Triggers è:

```
CREATE TRIGGER trigger_name
[STATUS {ACTIVE|INACTIVE}]
[PRIORITY key]
event_time event_type [event_target]
[IF condition]
EXECUTE [AFTER|DEFERRED] action[;]

event_time:
  BEFORE
  AFTER
  DEFERRED

event_type:
  DELETE
  STATEMENT DELETE
  UPDATE
  STATEMENT UPDATE
  INSERT
  STATEMENT INSERT
  ROLLBACK
  COMMIT

event_target:
  ON class_name
```



```
  ON class_name([CLASS] attribute_name)

condition:
  expression

action:
  REJECT
  INVALIDATE TRANSACTION
  PRINT message_string
  CALL statement
  INSERT statement
  UPDATE statement
  DELETE statement
  EVALUATE statement
```

Trigger STATUS Un Trigger può essere creato con uno stato attivo o inattivo. Se il Trigger è nello stato inattivo non può essere scatenato perchè l'evento a cui è associato non viene rilevato.

Trigger PRIORITY Esiste la possibilità che più di un Trigger alla volta venga scatenato, per assegnare un ordine preciso all'esecuzione delle regole si deve associare la priorità opportuna.

Trigger EVENT Ogni regola è associata con un singolo evento. Questo viene specificato da un tipo, un tempo e un target.

- *Tempo*: è equivalente alla proprità di *Coupling Mode* dell'evento spiegato del paragrafo 2.2.2. Quindi si avranno i valori di BEFORE, AFTER e DEFERRED.
- *Tipo*: ci sono due tipologie di evento principali in UNISQL:
  1. eventi relativi alle transazioni. Sono rilevati con un COMMIT o con un ROLLBACK. Questa tipologia non è contemplata dalla grammatica presente in questa tesi e quindi non viene oltremodo trattata.
  2. eventi relativi a modifiche. Sono rilevati con una richiesta di insert, update o delete. Per distinguere la granularotà, UNISQL usa le parole chiave INSERT, UPDATE, DELETE per le operazioni instance-oriented, mentre invece aggiunge la parola chiave STATEMENT per le operazioni set-oriented.
- *Target*: identifica l'oggetto sul quale si verifica l'evento. È dato come nome della classe con, eventualmente, il nome di un attributo.

obj	Si riferisce all'oggetto presente nel database. Può essere usato per accedere all'oggetto prima che questo sia modificato o cancellato. Può anche essere usato per identificare un oggetto subito dopo il suo inserimento
new	Si riferisce all'oggetto proposto per un inserimento o una modifica. Il nuovo oggetto può essere identificato solo prima che l'inserimento o la modifica siano effettuati
old	Si riferisce all'oggetto che è stato già modificato. Può essere utilizzato solo in combinazione con un UPDATE.

Tabella A.1: Funzioni degli identificatori

	BEFORE	AFTER or DEFERRED
INSERT	new	obj
UPDATE	obj, new	obj, old
DELETE	obj	NA

Tabella A.2: Utilizzo degli identificatori accoppiati al Coupling Mode

Trigger CONDITION La condizione è opzionale, se viene omessa il Trigger è detto incondizionato. Può essere scritto in differenti modi l'importante è che abbia come risultato un *true* o un *false*. È possibile fare riferimento agli oggetti anche attraverso gli identificatori *new, old, obj* (si veda tabella A.1). È possibile inserire anche una query con il costrutto *select-from-where* che viene utilizzato per reperire informazioni utili per un test di tipo booleano.

Trigger ACTION L'azione viene eseguita solo se la condizione viene verificata vera. Le azioni possono essere:

- REJECT: Non permette che la modifica al database che ha causato l'attivazione del Trigger sia portata a termine.
- INVALIDATE TRANSACTION: Causa il rollback dell'intera transazione. Impedisce che il database contenga dati incorretti.
- PRINT: Scrive a video il messaggio. È utile per funzioni di debug.
- INSERT: Permette l'inserimento di un oggetto nel database. La sintassi è la seguente:

```
INSERT INTO class_name [(attriubute_list)]
VALUES (value_list) [;]
```


calss_name è il nome della classe in cui viene effettuato l'inserimento, attribute_name è la lista degli attributi che si vogliono inserire e value_list è la lista dei valori da assegnare agli attributi.

- UPDATE: Permette la modifica degli attributi degli oggetti di una classe. La sintassi è:

```
UPDATE class_name
SET assignment [,assignment...]
[WHERE serch_condition] [;]

assignment: attribute_name={expression|NULL}
```

Esiste anche la possibilità di utilizzare anche un'altra forma che permette di modificare un oggetto solo.

```
UPDATE OBJECT obj_id
SET assignment [,assignment, ...] [;]
```

obj_id è l'identificatore dell'oggetto da modificare.

- DELETE: Un'istanza creata in una classe può essere cancellata. Con la clausola WHERE è possibile limitare i dati cancellati. La sintassi è:

```
DELETE
FROM class_name [correlation]
[WHERE search_condition] [:]

correlation: [AS] identifier
```

- CALL: Permette di invocare un metodo presente nel database. La sintassi è:

```
CALL method_call [;]

method_call:
 method_name([arg_value[,...]) ON obj_id [TO variable]
```

method_name si riferisce al nome del metodo da eseguire, mentre obj_id identifica l'oggetto su cui eseguire il metodo.

- EVALUATE: È uguale a CALL.

Appendice B

Lex & Yacc

Lex e *Yacc* sono due utility, molto usate in ambiente UNIX, 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 flusso 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 si riporta 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 ;
```

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

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

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.

[2]descritto ampiamente in [?]

Appendice C

Sintassi ODL

In questa appendice si trova la sintassi ODL riconosciuta dal traduttore. È espressa in formato Yacc, è stata estratta direttamente dal sorgente del programma.

```
%token <String_Type>
        Identifier
        String
        IntegerLiteral
        StringLiteral
        CharacterLiteral
        FloatingPtLiteral
%token <CommandType>
        SEMI
        LPAR
        RPAR
        MODULE
        DOUBLE_COLON
        COLON
        PERSISTENT
        TRANSIENT
        INTERFACE
        VIEW
        LRPAR
        RRPAR
        EXTENT
        KEY
        KEYS
        COMMA
        CONST
        EQUAL
        GREATEQUAL
        LESSEQUAL
        VERT
        HAT
        AMPER
        DOUBLE_RIGHT
        DOUBLE_LEFT
        PLUS
        MINUS
        TIMES
        SLASH
        PERCENT
        TILDE
        TRUE
        FALSE
        TYPEDEF
        FLOAT
        DOUBLE
        INT
        LONG
        SHORT
        UNSIGNED
        CHAR
        BOOLEAN
        ANY
        OCTET
        STRUCT
        UNION
        SWITCH
        CASE
        DEFAULT
        ENUM
        ARRAY
        LEFT
        RIGHT
        STRING
        LEPAR
        REPAR
        READONLY
        ATTRIBUTE
        SET
        LIST
        BAG
        INVERSE
        RELATIONSHIP
        ORDER_BY
        EXCEPTION
        ONEWAY
        VOID
        IN
        OUT
        INOUT
        RAISES
        CONTEXT
        INFINITE
        RANGE
        AND
        RULE
        THEN
        FOR
        ALL
        FORALL
        EXISTS
        DOT

%type <CommandType>
        ForAll

%type <Definition_list>
        Specification
        Definition
        Module

%type <Interface_type>
        Interface
        InterfaceDcl
        ForwardDcl

%type <Iner_list>
        InheritanceSpec

%type <Prop_list>
        OptInterfaceBody
        InterfaceBody
        Export

%type <Rela_type>
        RelDcl

%type <Attr_type>
        AttrDcl

%type <String_Type>
        AttributeName
        ScopedName
        BooleanLiteral
        ConstExp
        OrExpr
        AndExpr
        ShiftExpr
        AddExpr
        XOrExpr
        MultExpr
        UnaryExpr
        PrimaryExpr
        Literal
        PositiveIntConst
        EnumType
        TypeSpec
        SimpleTypeSpec
        CollectionType
        AttrCollectionSpecifier
        DomainType
        BaseTypeSpec
        FloatingPtType
        IntegerType
        SignedInt
        SignedLongInt
        SignedShortInt
        UnsignedInt
        UnsignedLongInt
        UnsignedShortInt
        SignedFloatingPtLiteral
        SignedIntegerLiteral
        IntegerValue
        LiteralValue
        CharType
        BooleanType
        OctetType
        AnyType
        TemplateTypeSpec
        ArrayType
```

```
        StringType
        ConstrTypeSpec
        StructType
        UnionType
        ArraySizeList
        FixedArraySize
        TraversalPathName1
        TraversalPathName2
        ConstType
        RangeType
        RangeSpecifier
        RuleConstOp
        RuleCast
        DottedName


%type <Character_Type>
        UnaryOperator
        OptOrderBy
        Signes

%type <Enum_enumerators_list>
        Enumerator
        EnumeratorList

%type <Struct_member_list>
        Member
        MemberList

%type <Declarator_type>
        Declarator
        SimpleDeclarator
        ComplexDeclarator
        ArrayDeclarator

%type <Declarator_list>
        Declarators

%type <Rela_target_type>
        TargetOfPath

%type <Rela_inverse_type>
         InverseTraversalPath

%type <Type_list>
        TypeDcl
        TypeDeclarator

%type <Const_type>
        ConstDcl

%type <Rule_type>
        RuleDcl

%type <Rule_body_list>
        RuleAntecedente
        RuleConseguente
        RuleBodyList
        RuleBody

%%

Odl_syntax:
        Specification

Specification:
        Definition
                        |
        Definition
        Specification

Definition:
        TypeDcl
        SEMI
                        |
        ConstDcl
        SEMI
                        |
        ExceptDcl
        SEMI
                        |
        Interface
        SEMI
                        |
        RuleDcl
        SEMI
                        |
        Module
        SEMI
                        |
        error
        SEMI

Module:
        MODULE
        Identifier
        LPAR
        Specification
```



```
        RPAR

Interface:
        InterfaceDcl
                        |
        ForwardDcl

InterfaceDcl:
        INTERFACE
        Identifier
        COLON
        InheritanceSpec
        OptTypePropertyList
        OptPersistenceDcl
        LPAR
        OptInterfaceBody
        RPAR
                        |
        INTERFACE
        Identifier
        OptTypePropertyList
        OptPersistenceDcl
        LPAR
        OptInterfaceBody
        RPAR
                        |
        VIEW
        Identifier
        COLON
        InheritanceSpec
        OptTypePropertyList
        OptPersistenceDcl
        LPAR
        OptInterfaceBody
        RPAR
                        |
        VIEW
        Identifier
        OptTypePropertyList
        OptPersistenceDcl
        LPAR
        OptInterfaceBody
        RPAR

ForwardDcl:
        INTERFACE
        Identifier

OptInterfaceBody:
        InterfaceBody


OptPersistenceDcl:
        |
        PersistenceDcl

PersistenceDcl:
        PERSISTENT
                        |
        TRANSIENT

OptTypePropertyList:
        |
        TypePropertyList

TypePropertyList:
        LRPAR
        OptExtentSpec
        OptKeySpec
        RRPAR

OptExtentSpec:
                        |
        ExtentSpec

ExtentSpec:
        EXTENT
        Identifier

OptKeySpec:
                        |
        KeySpec

KeySpec:
        KEY
        KeyList
                        |
        KEYS
        KeyList

KeyList:
        Key
        |
        Key
        COMMA
        KeyList
```

```
Key:
        PropertyName
                    |
        LRPAR
        PropertyList
        RRPAR

PropertyList:
        PropertyName
        |
        PropertyName
        COMMA
        PropertyList

PropertyName:
        Identifier

InterfaceBody:
        Export
                    |
        Export
        InterfaceBody

Export:
        TypeDcl
        SEMI
                    |
        ConstDcl
        SEMI
                    |
        ExceptDcl
        SEMI
                    |
        AttrDcl
        SEMI
                    |
        RelDcl
        SEMI
                    |
        OpDcl
        SEMI
                    |
        error
        SEMI

InheritanceSpec:
        ScopedName
                    |
        ScopedName
        COMMA
        InheritanceSpec

ScopedName:
        Identifier
                    |
        DOUBLE_COLON
        Identifier
                    |
        ScopedName
        DOUBLE_COLON
        Identifier

ConstDcl:
        CONST
        STRING
        Identifier
        EQUAL
        StringLiteral
                    |
        CONST
        CharType
        Identifier
        EQUAL
        CharacterLiteral
                    |
        CONST
        IntegerType
        Identifier
        EQUAL
        SignedIntegerLiteral
                    |
        CONST
        FloatingPtType
        Identifier
        EQUAL
        SignedFloatingPtLiteral

SignedIntegerLiteral:
        Signes
        IntegerLiteral
                    |
        IntegerLiteral


SignedFloatingPtLiteral:
        Signes
        FloatingPtLiteral
```



```
                    |
        FloatingPtLiteral

Signes:
        MINUS
                    |
        PLUS

ConstType:
        IntegerType
                    |
        CharType
                    |
        BooleanType
                    |
        FloatingPtType
                    |
        StringType
                    |
        ScopedName

ConstExp:
        OrExpr


OrExpr:
        XOrExpr
                    |
        OrExpr
        VERT
        XOrExpr

XOrExpr:
        AndExpr
                    |
        XOrExpr
        HAT
        AndExpr

AndExpr:
        ShiftExpr
                    |
        AndExpr
        AMPER
        ShiftExpr

ShiftExpr:
        AddExpr
                    |
        ShiftExpr
        DOUBLE_RIGHT
        AddExpr
                    |
        ShiftExpr
        DOUBLE_LEFT
        AddExpr

AddExpr:
        MultExpr
                    |
        AddExpr
        PLUS
        MultExpr
                    |
        AddExpr
        MINUS
        MultExpr

MultExpr:
        UnaryExpr
                    |
        MultExpr
        TIMES
        UnaryExpr
                    |
        MultExpr
        SLASH
        UnaryExpr
                    |
        MultExpr
        PERCENT
        UnaryExpr

UnaryExpr:
        UnaryOperator
        PrimaryExpr
                    |
        PrimaryExpr

UnaryOperator:
        MINUS
                    |
        PLUS
                    |
        TILDE

PrimaryExpr:
        ScopedName
```

```
                |
        Literal
                |
        LRPAR
        ConstExp
        RRPAR
                |
        LRPAR
        error
        RRPAR

Literal:
        IntegerLiteral
                |
        StringLiteral
                |
        CharacterLiteral
                |
        FloatingPtLiteral
                |
        BooleanLiteral

BooleanLiteral:
        TRUE
                |
        FALSE

PositiveIntConst:
        ConstExp

TypeDcl:
        TYPEDEF
        TypeDeclarator
                |
        StructType
                |
        UnionType
                |
        EnumType

TypeDeclarator:
        SimpleTypeSpec
        Declarators
                |
        ConstrTypeSpec
        Declarators

TypeSpec:
        SimpleTypeSpec
                |
        ConstrTypeSpec

SimpleTypeSpec:
        BaseTypeSpec
                |
        TemplateTypeSpec
                |
        ScopedName

BaseTypeSpec:
        FloatingPtType
                |
        IntegerType
                |
        CharType
                |
        BooleanType
                |
        OctetType
                |
        RangeType
                |
        AnyType


TemplateTypeSpec:
        ArrayType
                |
        StringType
                |
        CollectionType

ConstrTypeSpec:
        StructType
        |
        UnionType
        |
        EnumType

Declarators:
        Declarator
                |
        Declarator
        COMMA
        Declarators

Declarator:
        SimpleDeclarator
```



```
                |
        ComplexDeclarator

SimpleDeclarator:
        Identifier

ComplexDeclarator:
        ArrayDeclarator

FloatingPtType:
        FLOAT
                |
        DOUBLE
           RangeType:
        RANGE
        LPAR
        RangeSpecifier
        RPAR

RangeSpecifier:
        IntegerValue
        COMMA
        IntegerValue
                |
        IntegerValue
        COMMA
        PLUS
        INFINITE
                |
        MINUS
        INFINITE
        COMMA
        IntegerValue

IntegerValue:
        SignedIntegerLiteral
                |
        Identifier


IntegerType:
        INT
                |
        SignedInt
                |
        UnsignedInt

SignedInt:
        SignedLongInt
                |
        SignedShortInt

SignedLongInt:
        LONG

SignedShortInt:
        SHORT

UnsignedInt:
        UnsignedLongInt
                |
        UnsignedShortInt

UnsignedLongInt:
        UNSIGNED
        LONG

UnsignedShortInt:
        UNSIGNED
        SHORT

CharType:
        CHAR

BooleanType:
        BOOLEAN

OctetType:
        OCTET

AnyType:
        ANY

StructType:
        STRUCT
        Identifier
        LPAR
        MemberList
        RPAR

MemberList:
        Member
                |
        Member
        MemberList

Member:
        TypeSpec
```

```
        Declarators
        SEMI

UnionType:
        UNION
        Identifier
        SWITCH
        LRPAR
        SwitchTypeSpec
        RRPAR
        LPAR
        SwitchBody
        RPAR

SwitchTypeSpec:
        IntegerType
                |
        CharType
                |
        BooleanType
                |
        EnumType
                |
        ScopedName
                |
        RangeType

SwitchBody:
        Case
        |
        Case
        SwitchBody

Case:   CaseLabelList
        ElementSpec
        SEMI

CaseLabelList:
        CaseLabel
        |
        CaseLabel
        CaseLabelList

CaseLabel:
        CASE
        ConstExp
        COLON
                |
        DEFAULT
        COLON

ElementSpec:
        TypeSpec
        Declarator

EnumType:
        ENUM
        Identifier
        LPAR
        EnumeratorList
        RPAR

EnumeratorList:
        Enumerator
                |
        Enumerator
        COMMA
        EnumeratorList

Enumerator:
        Identifier

ArrayType:
        ARRAY
        LEFT
        SimpleTypeSpec
        COMMA
        PositiveIntConst
        RIGHT
                |
        ARRAY
        LEFT
        SimpleTypeSpec
        RIGHT

StringType:
        STRING
        LEFT
        PositiveIntConst
        RIGHT
                |
        STRING

ArrayDeclarator:
        Identifier
        ArraySizeList

ArraySizeList:
```



```
        FixedArraySize
                |
        FixedArraySize
        ArraySizeList

FixedArraySize:
        LEPAR
        PositiveIntConst
        REPAR

AttrDcl:
        READONLY
        ATTRIBUTE
        DomainType
        FixedArraySize
        AttributeName
                |
        ATTRIBUTE
        DomainType
        FixedArraySize
        AttributeName
                |
        READONLY
        ATTRIBUTE
        DomainType
        AttributeName
                |
        ATTRIBUTE
        DomainType
        AttributeName

AttributeName:
        Identifier

DomainType:
        SimpleTypeSpec
                |
        StructType
                |
        EnumType

CollectionType:
        AttrCollectionSpecifier
        LEFT
        Literal
        RIGHT
                |
        AttrCollectionSpecifier
        LEFT
        Identifier
        RIGHT
                |
        AttrCollectionSpecifier
        LEFT
        SimpleTypeSpec
        RIGHT

AttrCollectionSpecifier:
        SET
                |
        LIST
                |
        BAG
                |
        ARRAY

RelDcl:
        RELATIONSHIP
        TargetOfPath
        TraversalPathName1
        INVERSE
        InverseTraversalPath
        OptOrderBy

OptOrderBy:
        LPAR
        ORDER_BY
        AttributeList
        RPAR
                |

TraversalPathName1:
        Identifier

TargetOfPath:
        Identifier
                |
        RelCollectionType
        LEFT
        Identifier
        RIGHT

InverseTraversalPath:
        Identifier
        DOUBLE_COLON
        TraversalPathName2

TraversalPathName2:
```

```
        Identifier

AttributeList:
        ScopedName
                  |
        ScopedName
        COMMA
        AttributeList

RelCollectionType:
        SET
                  |
        LIST

ExceptDcl:
        EXCEPTION
        Identifier
        LPAR
        OptMemberList
        RPAR

OptMemberList:
        |
        MemberList

OpDcl:
        OpAttribute
        OpTypeSpec
        Identifier
        ParameterDcls
        RaisesExpr
        ContextExpr
                  |
        OpTypeSpec
        Identifier
        ParameterDcls
        RaisesExpr
        ContextExpr
                  |
        OpAttribute
        OpTypeSpec
        Identifier
        ParameterDcls
        ContextExpr
                  |
        OpAttribute
        OpTypeSpec
        Identifier
        ParameterDcls
```

```
        RaisesExpr
                  |
        OpTypeSpec
        Identifier
        ParameterDcls
        ContextExpr
                  |
        OpAttribute
        OpTypeSpec
        Identifier
        ParameterDcls
                  |
        OpTypeSpec
        Identifier
        ParameterDcls
        RaisesExpr
                  |
        OpTypeSpec
        Identifier
        ParameterDcls

OpAttribute:
        ONEWAY

OpTypeSpec:
        SimpleTypeSpec
                  |
        VOID

ParameterDcls:
        LRPAR
        ParamDclList
        RRPAR
                  |
        LRPAR
        RRPAR

ParamDclList:
        ParamDcl
        |
        ParamDcl
        COMMA
        ParamDclList

ParamDcl:
        ParamAttribute
        SimpleTypeSpec
        Declarator
                  |
```



```
        ParamAttribute
        AttrCollectionSpecifier
        LEFT
        SimpleTypeSpec
        RIGHT
        Declarator

ParamAttribute:
        IN
                  |
        OUT
                  |
        INOUT

RaisesExpr:
        RAISES
        LRPAR
        ScopedNameList
        RRPAR

ScopedNameList:
        ScopedName
                  |
        ScopedName
                COMMA
        ScopedNameList

ContextExpr:
        CONTEXT
        LRPAR
        StringLiteralList
        RRPAR

StringLiteralList:
        StringLiteral
                  |
        StringLiteral
        COMMA
        StringLiteralList

RuleDcl:
        RULE
        Identifier
        RuleAntecedente
        THEN
        RuleConseguente

RuleAntecedente:
        ForAll
```

```
        Identifier
        IN
        Identifier
        COLON
        RuleBodyList

RuleConseguente:
        RuleBodyList

RuleBodyList:
        LRPAR
        RuleBodyList
        RRPAR
                |
        RuleBody
                |
        RuleBodyList
        AND
        RuleBody
                |
        RuleBodyList
        AND
        LRPAR
        RuleBodyList
        RRPAR

LiteralValue:
        SignedFloatingPtLiteral
        |
        SignedIntegerLiteral
        |
        CharacterLiteral
        |
        StringLiteral
        |
        Identifier

RuleBody:
        DottedName
        RuleConstOp
        LiteralValue
                |
        DottedName
        RuleConstOp
        RuleCast
        LiteralValue
                |
        DottedName
        IN
```

```
        SimpleTypeSpec
                |
        ForAll
        Identifier
        IN
        DottedName
        COLON
        RuleBodyList
                |
        EXISTS
        Identifier
        IN
        DottedName
        COLON
        RuleBodyList

FunctionDef:
        Identifier
        LRPAR
        DottedNameList
        RRPAR
        |
        Identifier
        LRPAR
        RRPAR

DottedNameList:
        DottedName
        |
        LiteralValue
        |
        DottedName
        COMMA
        DottedNameList
        |
        LiteralValue
        COMMA
        DottedNameList
        |
        SimpleTypeSpec
        DottedName
        |
        SimpleTypeSpec
        LiteralValue
        |
        SimpleTypeSpec
        DottedName
        COMMA
        DottedNameList
        |
        SimpleTypeSpec
        LiteralValue
        COMMA
        DottedNameList

RuleConstOp:
        EQUAL
                |
        GREATEQUAL
                |
        LESSEQUAL
                |
        LEFT
                |
        RIGHT


RuleCast:
        LRPAR
        SimpleTypeSpec
        RRPAR

DottedName:
        Identifier
                |
        Identifier
        DOT
        DottedName

ForAll:
        FOR
        ALL
        |
        FORALL

EventDcl:
        DEFINE
        EVENT
        Identifier
        EventDef

EventDef:
        MethodEvent
        |
        TimeEvent

MethodEvent:
           BEFORE
```



```
           Identifier
           DOT
           FunctionDef
           |
           AFTER
           Identifier
           DOT
           FunctionDef
TimeEvent:
         EVERY
         IntegerLiteral
         Cadency
         |
         EVERY
         IntegerLiteral
         Cadency
         AT
         IntegerLiteral
         COLON
         IntegerLiteral
          |
          EVERY
          IntegerLiteral
          Cadency
          AT
          IntegerLiteral
          COMMA
          IntegerLiteral

SysEvent:
          INSERT
          IN
          Identifier
          |
          DELETE
          IN
          Identifier
          |
          UPDATE
          IN
          Identifier

Cadency:
        YEAR
        |
        MONTH
        |
        DAY
        |
        HOUR
        |
        MINUTE

EcaRuleDcl:
         ECARULE
         Identifier
         ON
         Identifier
         IF
         Condition
         DO
         Action
         OptionList
         |
         ECARULE
         Identifier
         ON
         SysEvent
         IF
         Condition
         DO
         Action
         OptionList

Condition:
         TRUE
         |
         RuleBody
         |
         Identifier
         DOT
         FunctionDef
         |
         LEPAR
         SELECT
         QueryList
         REPAR
         RuleConstOp
         LiteralValue

QueryList:
         Query
         |
         Query
         QueryList

Query:
         DottedName
```

```
        |
        Identifier
        |
        Special
        |
        RuleConstOp
        LiteralValue

Special:
        SELECT
        |
        LRPAR
        |
        RRPAR
        |
        TIMES
        |
        COLON
        |
        AND
        |
        FOR
        |
        ALL
        |
        FORALL
        |
        EXISTS
        |
        IN
        |
        COMMA


Action:
        RuleBody
        |
        Identifier
        DOT
        FunctionDef
        |
        Identifier
        DOT
        FunctionDef
        AND
        Action
        |
        RuleBody
        AND
        Action
        |
        ABORT
         |
         Identifier
         DOT
         DELETE
         |
         DottedName
         EQUAL
         DottedName
         EcaOp
         LiteralValue

EcaOp:
         MINUS
         |
         PLUS
         |
         SLASH
         |
         TIMES


OptionList:
         /* Nessuna opzione */
         |
         Option
         |
         Option
         COMMA
         OptionList

Option:
         GRANULARITY
         EQUAL
         GranOpt
         |
         EC
         EQUAL
         CouplingOpt
         |
         AC
         EQUAL
         CouplingOpt
         |
         ATOMIC
         EQUAL
```



```
        AtomicOpt
        |
        CONSUMPTION
        SCOPE
        EQUAL
        ConsScopeOpt
        |
        CONSUMPTION
        TIME
        EQUAL
        ConsTimeOpt
        |
        PRECEDES
        LRPAR
        NameList
        RRPAR
        |
        FOLLOWS
        LRPAR
        NameList
        RRPAR

NameList:
        Identifier
        |
        Identifier
        COMMA
        NameList

AtomicOpt:
         ATOMIC
         |
         NO

ConsTimeOpt:
         CONDITION
         |
         EXECUTION

ConsScopeOpt:
         LOCAL
         |
         GLOBAL
         |
         NO

CouplingOpt:
         IMMEDIATE
         |
         DELAYED
         |
         DEFERRED

GranOpt:
         INSTANCE
         |
         SET
```

Appendice D

Sintassi OCDL

In questa sezione è riportata l'ultima una versione disponibile della sintassi[1] dell'OCDL riconosciuta dal validatore.

$$
\begin{array}{rcl}
< \textit{linguaggio} >< \text{linguaggio} > & ::= & < \text{def-term} > \mid \\
 & & < \text{linguaggio} >< \text{def-term} > \\
< \text{def-term} > & ::= & < \text{def-tipovalore} > \mid \\
 & & < \text{def-classe} > \mid \\
 & & < \text{def-regola} > \\
< \text{def-tipovalore} > & ::= & < \text{def-tipobase} > \mid \\
 & & < \text{def-tipo} > \\
< \text{def-classe} > & ::= & < \text{def-classe-prim} > \mid \\
 & & < \text{def-classe-virt} > \\
< \text{def-regola} > & ::= & < \text{def-antconV} > \\
 & & < \textit{nome regola} > = < \text{classe} > \mid \\
 & & < \text{def-antconT} > \\
 & & < \textit{nome regola} > = < \text{def-tipoT} > \\
< \text{def-antconV} > & ::= & \textbf{antev} \mid \textbf{consv} \\
< \text{def-antconT} > & ::= & \textbf{antet} \mid \textbf{const} \\
< \text{def-tipoT} > & ::= & < \text{tipo} > \mid < \text{tipobase} > \\
< \text{def-tipo} > & ::= & \textbf{type} < \textit{nome tipovalore} > = < \text{tipo} >
\end{array}
$$

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

$$
\begin{array}{rcl}
< \text{def-classe-prim} > & ::= & \textbf{prim} < \textit{nome classe} > = < \text{classe} > \\
< \text{def-classe-virt} > & ::= & \textbf{virt} < \textit{nome classe} > = < \text{classe} > \\
< \text{def-antecedente} > & ::= & < \text{tipobase} > \mid \\
 & & < \text{tipo} > \mid \\
 & & < \text{classe} > \\
< \text{def-conseguente} > & ::= & < \text{tipobase} > \mid \\
 & & < \text{tipo} > \mid \\
 & & < \text{classe} > \\
< \text{tipo} > & ::= & \#\textbf{top}\# \mid \\
 & & < \text{insiemi-di-tipi} > \mid \\
 & & < \text{esiste-insiemi-di-tipi} > \mid \\
 & & < \text{sequenze-di-tipi} > \mid \\
 & & < \text{ennuple} > \mid \\
 & & < \text{nomi-di-tipi} > \mid \\
 & & < \text{tipo-cammino} > \\
< \text{classe} > & ::= & < \text{insiemi-di-classi} > \mid \\
 & & < \text{esiste-insiemi-di-classi} > \mid \\
 & & < \text{sequenze-di-classi} > \mid \\
 & & < \text{nomi-di-classi} > \mid \\
 & & \hat{} < \text{tipo} > \mid \\
 & & < \text{nomi-di-classi} > \ \& \hat{} < \text{tipo} > \\
< \text{insiemi-di-tipi} > & ::= & \{ < \text{tipo} > \} \mid \\
 & & \{ < \text{tipo} > \} \ \& \ < \text{insiemi-di-tipi} > \mid \\
 & & \{ < \text{tipobase} > \} \\
< \text{esiste-insiemi-di-tipi} > & ::= & !\{ < \text{tipo} > \} \mid \\
 & & !\{ < \text{tipo} > \} \ \& \ < \text{esiste-insiemi-di-tipi} > \mid \\
 & & !\{ < \text{tipobase} > \} \\
< \text{sequenze-di-tipi} > & ::= & \langle \ < \text{tipo} > \ \rangle \mid \\
 & & \langle \ < \text{tipo} > \ \rangle \ \& \ < \text{sequenze-di-tipi} > \mid \\
 & & \langle \ < \text{tipobase} > \ \rangle \\
< \text{ennuple} > & ::= & [< \text{attributi} >] \mid \\
 & & [< \text{attributi} >] \ \& \ < \text{ennuple} > \\
< \text{attributi} > & ::= & < \textit{nome attributo} > : < \text{desc-att} > \mid \\
 & & < \textit{nome attributo} > : < \text{desc-att} > , < \text{attributi} >
\end{array}
$$

< desc-att > ::= < tipobase > |
< tipo > |
< classe >

< nomi-di-tipi > ::= < *nome tipovalore* > |
< *nome tipovalore* > & < nomi-di-tipi >

< tipo-cammino > ::= (< *nome attributo* > : < desc-att >)

< insiemi-di-classi > ::= { < classe > } |
{ < classe > } & < insiemi-di-classi >

< esiste-insiemi-di-classi > ::= !{ < classe > } |
!{ < classe > } & < esiste-insiemi-di-classe >

< sequenze-di-classi > ::= ⟨ < classe > ⟩
⟨ < classe > ⟩ & < sequenze-di-classi >

< nomi-di-classi > ::= < *nome classe* > |
< *nome classe* > & < nomi-di-classi >

< def-tipobase > ::= **btype** < *nome tipobase* > = < tipobase >

< tipobase > ::= **real** |
integer |
string |
boolean |
< range-intero > |
vreal < *valore reale* > |
vinteger < *valore intero* > |
vstring < *valore string* > |
vboolean < valore-boolean > |
< *nome tipobase* >

< valore boolean > ::= **true** | **false**

< range-intero > ::= **range** < *valore intero* > +**inf** |
−**inf** < *valore intero* > |
< *valore intero* > < *valore intero* >

Appendice E

File di output del traduttore per lo schema Clinic

```
clinic.sc
type Address_s = [  street : string , city : string ,
                    tel_number : string ] ;
prim Room =  ^ [ address : Address_s , number : integer ,
                 req : { Request }  ] ;
prim To_day =  Request ;
prim Request = Visit &^[ date_req : string,date_fixed : string ,
                         ticket : integer ,  by : Person ,
                         in_the : Room  ] ;
prim Missed = Visit &^[ motivation : string ] ;
prim Doctor = ^ [ name1 : string , address : Address_s ,
                  specialization : string ,  doc : { Visit } ] ;
prim Visit = ^ [ number : integer, type1: string, doct : Doctor,
                 inc : Clinical_folder  , result : string ,
                 state : string ] ;
prim Clinical_folder = ^ [number : integer, last_update : bool,
                          general_status : string ,  of : Person ,
                          with : { Visit }  ] ;
prim Person = ^ [name1 : string , address : Address_s ,
                 requested : { Request }, have : Clinical_folder,
                   number_card : integer , state : string ] ;
operation Request = top f print1 ( in num_copies : integer ) ;
operation Request = integer f cl_ticket ( in n_c : integer ,
                           in c_c : integer ) ;
operation Doctor = top f salary ( ) ;
operation Visit = top f print1 ( ) ;
operation Visit = string f assign ( in type1 : string ) ;
```



```
operation Visit = bool f n_esente ( in n_c : integer ) ;
operation Visit = bool f n_payed ( ) ;
operation Visit = string f today ( ) ;
operation Visit = top f show ( ) ;
operation Clinical_folder = top f send (in address :string);
operation Clinical_folder = bool f ask ( ) ;
operation Person = integer f age ( ) ;
operation Person = top f print1 ( ) ;
operation Person = top f file1 ( ) ;
virt finish1a = Request & ^ [state : vstring "not_executed"];
virt finisha = Request & ^ [state : vstring "executed" ] ;
virt starta = Request & ^ [date_fixed: string function today()];
evedef b_send m b Clinical_folder send ( ) ;
evedef evening t 1 d 21 : 00  ;
evedef morning t 1 d 8 : 00  ;
evedef show m b Visit show ( )  ;
ecarule send_info on b_send
      if c O.general_status = "private" do e O ask ( ) ;
ecarule print1 evedef i Request
      if t do e NEW print1 ( in 1 : integer )
             opt g I/O , f ( ticket )  ;
ecarule ticket evedef i Request
      if e NEW n_esente ( in NEW.by.number_card : integer )
       do i NEW.ticket = integer cl_ticket (in NEW.by.number_card:
                integer , in NEW.by.number_card : integer )
            opt g I/O  ;
ecarule archi1 evedef d Clinical_folder
      if c OLD.of.state = "alive" do a  ;
ecarule archi evedef u Person
      if c NEW.state = "transfer" do e NEW file1() and d NEW;
ecarule finish1 on evening
      if f X Request : c X.state ="not_executed"
        do i X in Missed and d X  ;
ecarule finish on evening
      if f X Request : c X.state = "executed" do d X  ;
ecarule start on morning
      if f X Request : i X.date_fixed = string today ( )
         do i X in To_day  ;
ecarule pr_request evedef i Request
      if t do e NEW print1 ( in 1 : integer )  ;
ecarule show_eca on show if e O n_payed ( ) do a  ;
ecarule in_person evedef i Person if c NEW.name1 = "Andrea" do a  .
```