

ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

FACOLTA' DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

DEIS

TESI DI LAUREA

in
Fondamenti di Informatica L-B

**Progetto e sviluppo di una libreria
tuProlog per il linguaggio SQL**

CANDIDATO:
Matteo Iuliani

RELATORE:
Chiar.mo Prof. Enrico Denti

CORRELATORE:
Dott. Ing. Giulio Piancastelli

Anno Accademico 2006/2007
Sessione II

Sommario

Introduzione	4
1 Modelli e tecnologie.....	7
1.1 Database e modello relazionale.....	7
1.2 Prolog e programmazione logica	8
1.3 Mapping tra i due modelli.....	10
1.4 Il motore tuProlog	11
1.5 Le librerie tuProlog	12
2 Analisi dello stato dell'arte.....	13
2.1 La libreria di CIAO Prolog	13
2.1.1 SQL persistent database interface	14
2.1.2 Prolog to SQL translator	15
2.1.3 Low-level socket interface to SQL/OBDC databases.....	16
2.1.4 Altre librerie	17
2.2 Prolog to SQL translator	17
2.2.1 Il predicato <code>translate/3</code>	17
2.2.2 Il predicato <code>printqueries/1</code>	19
2.2.3 Mapping di tabelle e colonne	19
2.2.4 Esempi di utilizzo del translator	20

3 Realizzazione della libreria	24
3.1 Sviluppo di nuove librerie tuProlog.....	24
3.2 Progetto	25
3.2.1 Utilizzo di JDBC	26
3.2.2 API e translator	27
3.2.3 Modifiche da apportare al translator	27
3.2.4 Gestione dei tipi di dato	27
3.2.5 Campi di istanza e altre utilità.....	28
3.2.6 Diagramma UML	28
3.3 Sviluppo	28
3.3.1 Campi di istanza.....	28
3.3.2 Direttive	30
3.3.3 Predicati	30
3.3.4 Metodi privati.....	33
3.3.5 Il metodo <code>String getTheory()</code>	34
3.4 Collaudo	36
3.4.1 La classe astratta <code>SQLLibraryTestCase</code>	37
3.4.2 La classe <code>SQLLibraryMySQLTestCase</code>	39
 4 Manuale di utilizzo	 44
4.1 Operazioni preliminari	44
4.1.1 Caricamento della libreria	44
4.1.2 Mapping di tabelle e colonne del database	46

4.1.3 Driver JDBC	47
4.2 Predicati.....	48
4.3 Esempi di utilizzo dei predicati.....	51
4.3.1 dbassert/1.....	51
4.3.2 dbretract/1	52
4.3.3 dbcurrent/1	53
4.3.4 dbretractall/1.....	54
4.3.5 dbfindall/3	54
4.3.6 get_tables/1.....	56
4.3.7 table_types/3	56
 Conclusioni	 57
 Bibliografia	 58

Introduzione

Tra i vari modelli di rappresentazione dei dati, il più diffuso (e implementato sulla maggior parte dei sistemi di gestione di basi di dati) è sicuramente il modello relazionale, che rappresenta i dati attraverso relazioni espresse in forma tabellare. Le righe di ogni tabella (dette anche tuple) costituiscono un insieme di dati che soddisfano la relazione.

Il linguaggio utilizzato per gestire un database e le sue tabelle è tipicamente SQL, un linguaggio di programmazione dichiarativo in cui il programmatore non indica come eseguire una certa operazione (ad esempio una query), ma solo quale operazione eseguire ed eventualmente la forma del risultato. Esistono inoltre operazioni per inserire e rimuovere tuple di dati.

Assai simile è ciò che accade nel linguaggio logico Prolog: anch'esso adotta uno stile dichiarativo, con programmi costituiti da un insieme di clausole, alcune delle quali (i fatti) esprimono una relazione. Pertanto un insieme di fatti costituisce un database. Anche Prolog permette di eseguire query sul database, sia di aggiornarlo.

Date le similitudini tra i due linguaggi, è interessante valutare se e come si possa accedere da programmi Prolog a database relazionali: in questo modo la base di conoscenza di un programma Prolog non sarebbe costituita soltanto dai fatti presenti nel programma, ma anche dalle tuple memorizzate persistentemente in un database relazionale.

tuProlog è un motore Prolog leggero e basato su Java, e le sue caratteristiche principali sono la minimalità e la configurabilità. Infatti il cuore del motore tuProlog è un oggetto Java che contiene soltanto le proprietà essenziali di un motore Prolog; ulteriori funzionalità possono essere aggiunte o rimosse dal motore in base alle necessità attraverso il meccanismo delle librerie. Tra le librerie standard è particolarmente importante la `JavaLibrary`, che permette di usare qualunque oggetto Java da programmi tuProlog.

In particolar modo è quindi possibile utilizzare le classi che costituiscono la tecnologia JDBC (un insieme di API che permette l'accesso universale ai database da programmi Java) e accedere in tal modo a database relazionali. Tuttavia, nonostante l'astrazione offerta da JDBC, l'utilizzo diretto dei predicati della `JavaLibrary` costringerebbe l'utente ad avere una visione troppo di “basso livello” del problema: dovrebbe conoscere abbastanza bene la tecnologia JDBC, dovrebbe eseguire interrogazioni al database attraverso SQL e dovrebbe maneggiare direttamente gli oggetti del package `java.sql`.

L'obiettivo di questa tesi è realizzare una libreria di predicati che consenta l'accesso a database relazionali da programmi tuProlog in modo diretto e decisamente più espressivo e naturale rispetto a quanto risulterebbe dal passaggio attraverso i predicati della `JavaLibrary`.

Infatti la libreria che è oggetto della tesi intende aggiungere un nuovo livello di astrazione sopra JDBC, consentendo all'utente di accedere a database relazionali da programmi tuProlog senza utilizzare direttamente oggetti Java oppure eseguire query SQL. I predicati della libreria utilizzeranno naturalmente JDBC, ma tutto avverrà nella maniera più trasparente. Le query, invece, dovranno essere espresse in Prolog: questo permette anche di mantenere una certa uniformità a livello di linguaggio.

Per raggiungere tali obiettivi, in fase di sviluppo della libreria è stato dapprima necessario eseguire un mapping del linguaggio SQL su termini e predicati Prolog, poi si è utilizzato JDBC per interfacciarsi verso le basi di dati.

La tesi è quindi strutturata nel modo seguente:

- il capitolo 1 ha lo scopo di presentare i modelli e le tecnologie cui si fa riferimento nel corso della trattazione, introducendo in questo modo anche la terminologia necessaria;

- nel capitolo 2 viene eseguita un'analisi delle scelte di modello e interfaccia adottate nel motore CIAO Prolog, ed è analizzato il funzionamento del Prolog to SQL translator, uno strumento già esistente per mappare il linguaggio SQL su termini e predicati Prolog;
- il capitolo 3 ripercorre le fasi della realizzazione della libreria: progetto, sviluppo e collaudo;
- infine nel capitolo 4 il lettore può trovare il manuale della libreria, con la descrizione e alcuni esempi di utilizzo dei predicati messi a disposizione.

Capitolo 1

Modelli e tecnologie

Il capitolo presenta i modelli e le tecnologie cui si fa riferimento nel seguito della trattazione, introducendo in questo modo anche la terminologia necessaria.

1.1 Database e modello relazionale

Il modello relazionale è un modello logico di rappresentazione dei dati implementato sulla maggior parte sistemi di gestione di basi di dati (DBMS), detti perciò sistemi di gestione di basi di dati relazionali (RDBMS).

L'assunto fondamentale del modello relazionale è che tutti i dati sono rappresentati come relazioni. La consistenza dei dati è ottenuta attraverso la specifica di appropriati *vincoli*.

Nel modello relazionale il *dominio* (o tipo di dato) è definito come l'insieme dei valori che può assumere un determinato attributo. Una *tupla* è un insieme non ordinato di valori degli attributi. Un *attributo* è una coppia ordinata <nome di attributo, nome di tipo>, mentre un valore di attributo è un valore specifico valido per quel tipo di dato. Una *relazione* consiste di una testata e di un corpo, dove la testata è un insieme di attributi e il corpo è un insieme tuple. La testata di una relazione è anche la testata di ciascuna delle sue tuple. La tabella è la rappresentazione grafica normalmente accettata per rappresentare la relazione. La consistenza dei dati in una tabella o tra più tabelle può essere ottenuta tramite vincoli, come ad esempio: vincoli di dominio (si riferiscono ai valori ammissibili per un singolo attributo), vincoli di tupla (esprimono condizioni su ciascuna tupla), vincoli di chiave (vietano la presenza di tuple distinte che hanno lo stesso valore su

uno o più attributi), vincoli di integrità referenziale (regolano le correlazioni di tuple appartenenti a relazioni diverse).

Il principio base del modello relazionale è che tutte le informazioni siano rappresentate da valori inseriti in relazioni (tabelle); dunque un database relazionale è un insieme di relazioni contenenti valori e il risultato di qualunque interrogazione (o manipolazione) dei dati può essere rappresentato anch'esso da relazioni (tabelle).

I *tipi di dato* utilizzati nei database relazionali possono essere numeri interi, caratteri alfanumerici, date, valori booleani, e così via. I corrispondenti “nomi di tipo” potranno essere le stringhe INT, CHAR, DATE, BOOLEAN, etc.

Attributo è il termine usato nella teoria per ciò che normalmente si definisce come *colonna*. Allo stesso modo *tabella* è normalmente usato al posto del termine teorico relazione. La struttura di una tabella è specificata come una lista di colonne, ciascuna delle quali ha un nome univoco e un dominio, un insieme cioè di valori accettati. Un valore di attributo è il valore di una cella identificata da una specifica coppia riga-colonna. Una tupla è in sostanza la stessa cosa di una *riga*. Una relazione è la definizione di una tabella (cioè un insieme di colonne) insieme ai dati che vi compaiono. La definizione della tabella è la testata e i dati che vi appaiono sono il corpo, un insieme di righe.

1.2 Prolog e programmazione logica

Prolog è un linguaggio di programmazione che adotta il paradigma di programmazione logica, che consiste nell'uso della logica matematica nella programmazione. Secondo questo paradigma la logica è usata come un linguaggio di rappresentazione puramente dichiarativo, e un dimostratore di teoremi si occupa della risoluzione del problema. Il compito di risolvere un problema è diviso tra il programmatore, che ha la responsabilità di assicurare la correttezza di un programma espresso in forma logica, e il

dimostratore di teoremi, che si occupa della risoluzione del problema in maniera efficiente.¹

La programmazione logica si basa quindi sull'applicazione di un dimostratore di teoremi a sentenze dichiarative espresse sotto forma di implicazioni. Ad esempio:

$$B_1 \text{ and } \dots \text{ and } B_n \text{ implica } H$$

diviene la seguente procedura:

per dimostrare/risolvere H , dimostra/risolvi B_1 e ... e B_n .

Un programma Prolog descrive relazioni, definite attraverso *clausole*. Queste possono essere di due tipi: *fatti* e *regole*. Una regola è della forma:

Head :- Body.

letta come “Head è vera se Body è vero”. Il corpo di una regola (Body) consiste di una serie di chiamate a *predicati*, denominati anche *goal* della regola. Clausole senza corpo sono chiamate fatti.²

L'esecuzione di un programma Prolog inizia con una query da parte dell'utente, che è una richiesta di dimostrazione. Il motore Prolog, utilizzando la sua base di conoscenza costituita da fatti e regole, cerca di confutare la query negata. Se la confutazione ha successo, segue che la query originale, con l'appropriata

¹Nella realtà non è proprio così, perché spesso il programmatore è costretto a lasciare da parte l'aspetto logico-dichiarativo del problema per intervenire sull'aspetto procedurale (demandato di solito al dimostratore) per motivi di efficienza. Tuttavia l'aspetto dichiarativo mantiene la sua importanza, anche come strumento per assicurare la correttezza di un programma.

²Le clausole senza corpo hanno in realtà il corpo uguale all'atomo `true`.

istanziamento delle variabili, è una conseguenza logica del programma. In questo caso la query ha successo, e tutte le istanziazioni di variabili valide sono riportate all'utente (attraverso un meccanismo chiamato *backtracking*). In caso contrario la query fallisce.

1.3 Mapping tra i due modelli

Un mapping tra i due modelli è possibile. Notiamo infatti che:

- sia nei database relazionali, sia nei programmi Prolog, i dati sono rappresentati sotto forma di relazioni. Nel primo caso le relazioni che intercorrono tra i dati sono espresse in forma tabellare, e le righe di una tabella costituiscono un insieme di dati che soddisfano una determinata relazione. Nel secondo caso invece le relazioni sono espresse attraverso un insieme di fatti. Pertanto sia un insieme di tabelle, sia un insieme di fatti Prolog, costituisce un database;
- SQL e Prolog sono due linguaggi dichiarativi: il programmatore non deve indicare come eseguire una certa operazione, ma deve solo specificare quale operazione eseguire ed eventualmente quale dovrà essere la forma del risultato;
- entrambi i linguaggi prevedono API di aggiornamento delle relazioni presenti nel database: in SQL esistono le istruzioni INSERT e DELETE, che servono per aggiungere e cancellare righe dalle tabelle del database; Prolog mette a disposizione i predicati `assert` e `retract`, che rendono possibile l'aggiornamento (aggiunta e rimozione di fatti) del database durante l'esecuzione di un programma.

Detto questo, e vista la similarità delle due situazioni, un mapping tra i due modelli sarebbe molto utile perché renderebbe possibile accedere da programmi Prolog a database relazionali. Tale possibilità darebbe principalmente due benefici:

- la base di conoscenza di un programma Prolog non sarebbe costituita soltanto dai fatti presenti nel programma, ma anche

dalle tuple memorizzate persistentemente in un database relazionale;

- più programmi Prolog potrebbero accedere contemporaneamente agli stessi dati presenti in un database, ed eventualmente modificarli.

La libreria che è oggetto della tesi si prefigge di realizzare questo mapping, rendendo così possibile l'interfacciamento verso le basi di dati. In tale situazione i predicati utili sarebbero sostanzialmente:

- una versione “persistente” dei predicati `assert` e `retract`, che permetta di inserire o cancellare tuple di un database relazionale;
- un predicato che permetta di “importare” nel motore Prolog i fatti di interesse presenti nel database, eseguendo una sorta di query (in linguaggio Prolog).

1.4 Il motore tuProlog

tuProlog è un motore Prolog leggero e basato su Java. Le sue caratteristiche principali sono:

- è facilmente *distribuibile*: tutto ciò che serve per la sua installazione sono la presenza di una macchina virtuale Java standard e un singolo file JAR.
- è *minimale*: il cuore del motore Prolog è un oggetto Java che contiene le proprietà essenziali di un motore Prolog; ulteriori funzionalità possono essere aggiunte o rimosse dal motore in base alle necessità.
- è *configurabile*: è presente un semplice e potente meccanismo per caricare dinamicamente nel motore tuProlog i predicati, i funtori e gli operatori necessari; questo avviene attraverso le *librerie* tuProlog.
- è *integrato con Java*: ogni entità Java (oggetto, classe, package) può essere rappresentata e usata come un termine Prolog; dall'altro lato uno o più motori Prolog possono essere invocati e usati come semplici oggetti Java.

- supporta l'*interoperabilità*: segue gli standard Internet, come ad esempio il supporto al protocollo TCP/IP e a RMI.

1.5 Le librerie tuProlog

Le librerie sono il mezzo attraverso il quale tuProlog raggiunge le sue caratteristiche fondamentali di minimalità e configurabilità. Per scelta progettuale il motore include pochi predicati built-in, che forniscono le funzionalità base del motore. Tuttavia altre funzionalità (predicati, funtori, operatori) sono fornite attraverso le librerie e possono essere aggiunte o sottratte al motore in ogni momento. Un motore tuProlog può quindi essere esteso dinamicamente caricando un numero qualsiasi di librerie. Ogni libreria fornisce un insieme di predicati e funtori e definisce anche un nuovo pezzo di teoria.

Nonostante il motore tuProlog includa soltanto le funzionalità essenziali, nella distribuzione sono incluse alcune librerie standard che sono caricate di default al momento dell'istanziamento del motore. Queste librerie forniscono predicati, funtori e operatori comuni a tutti i motori Prolog, il supporto alla Definite Clause Grammar, i predicati e i funtori presenti nella sezione built-in dello standard ISO, i predicati per l'I/O e i predicati per l'interoperabilità tra Java e Prolog. Possono poi essere caricate altre librerie definite dall'utente, come ad esempio quella per il linguaggio SQL che vogliamo realizzare e di cui abbiamo accennato nella sezione 1.3.

Capitolo 2

Analisi dello stato dell'arte

Analizzando la documentazione dei più importanti motori Prolog in circolazione emerge che soltanto CIAO Prolog implementa il concetto di libreria di predicati per l'accesso a database relazionali.

2.1 La libreria di CIAO Prolog

La libreria di Ciao Prolog implementa una versione “persistente” dei predicati built-in `assert` e `retract`, oltre a mettere a disposizione altri predicati utili per costruire viste arbitrariamente complesse delle relazioni presenti nel database sotto forma di predicati Prolog. Sono inoltre supportate le funzioni di aggregazione del linguaggio SQL ed è utilizzato un translator per generare dinamicamente codice SQL da query in linguaggio Prolog. La libreria è stata progettata per perseguire due finalità:

- semplificare il più possibile la comunicazione tra il motore Prolog e i database relazionali;
- dare la maggiore flessibilità possibile a tutto il motore, il che vuol dire permettere accessi simultanei a più database, con il client e la base dati che possono risiedere sulla stessa macchina o su macchine differenti, e il client può indifferentemente risiedere su una macchina Windows o Unix.

Per raggiungere tale flessibilità è utilizzata un'architettura client-server. Lato server un server MySQL si connette al database. Lato client l'interfaccia client di MySQL si connette a tale server. Naturalmente sulla macchina server deve girare il demone MySQL.

Dopo che la connessione è stata stabilita il client può inviare comandi al server mediatore, il quale li inoltrerà al database; i dati viaggeranno poi in senso opposto. I messaggi inviati da una parte e

dall'altra comprendono comandi di login e logout dal database, query SQL e risposte a tali query.

L'architettura complessiva è su due livelli:

- al più basso livello c'è l'interfaccia verso MySQL, che si occupa di inoltrare comandi al server MySQL e ricevere risposte;
- più in alto ci sono i predicati che usano il translator per costruire appropriati comandi SQL da goal Prolog, si servono del livello più basso per interfacciarsi con il database, interpretano le risposte e infine le presentano al motore Prolog attraverso il backtracking.

In realtà la libreria di CIAO Prolog non è unica; più esattamente si tratta di un insieme di librerie che consentono l'accesso a database relazionali. Queste sono:

- *SQL persistent database interface*;
- *Prolog to SQL translator*;
- *Low-level socket interface to SQL/ODBC databases*;
- *Types for the Low-level interface to SQL databases*;
- *sqltypes*.

Vediamo ora più precisamente la funzione e di ognuna di queste librerie e i principali predicati presenti.

2.1.1 SQL persistent database interface

Questa è la libreria di più alto livello, in altre parole quella utilizzata direttamente dagli utenti. I predicati principali sono:

- `dbassertz_fact(+Fact)`

L'istanza corrente di `Fact` è interpretata come una tupla di una relazione nel database ed è aggiunta alla tabella corrispondente al funtore principale di `Fact`.

- `dbretract_fact(+Fact)`

Cancella dal database un singolo fatto che unifica con **Fact**. Attraverso il backtracking è possibile cancellare dal database tutti i fatti che unificano con **Fact**.

- **dbcurrent_fact(+Fact)**
Verifica che nel database corrente esista un fatto che unifica con **Fact**. Attraverso il backtracking recupera dal database tutti i fatti che unificano con **Fact**.
- **dbretractall_fact(+Fact)**
Cancella dal database tutti i fatti che unificano con **Fact**.
- **dbfindall(+DBId, +Pattern, +ComplexGoal, -Results)**
Simile a **findall/3**, soltanto che **ComplexGoal** è eseguito sul database **DBId**. **Results** contiene tutte le tuple che rappresentano la risposta dal database **DBId** alla query Prolog **ComplexGoal**, proiettate su **Pattern** (ogni tupla presente in **Results** deve seguire il pattern indicato da **Pattern**). Usa il translator per generare la query SQL partendo da **ComplexGoal**.
- **sql_get_tables(+Location, -Tables)**
Tables contiene le tabelle disponibili in **Location**.
- **sql_table_types(+Location, +Table, -AttrTypes)**
AttrTypes contiene i nomi e i tipi di dato delle colonne della tabella **Table** in **Location**.

2.1.2 Prolog to SQL translator

Questa libreria si occupa di tradurre le query Prolog in codice SQL.

Il predicato principale è:

- `pl2sqlstring(+ProjectionTerm, +DatabaseGoal, -SQLQueryString)`

Traduce complessi goal Prolog (`DatabaseGoal`) nel corrispondente codice SQL. Il codice è preparato in modo che i risultati dell'esecuzione della query SQL siano proiettati sul termine `ProjectionTerm` (come nel caso del primo argomento di `findall/3`). `SQLQueryString` contiene il codice della query SQL, pronto per essere inviato a un server MySQL.

Nonostante questo predicato possa essere usato direttamente nei programmi, è più conveniente usare un livello più alto di astrazione: quello rappresentato dai predicati della *SQL persistent database interface*. In questo modo c'è un'interfaccia completamente trasparente tra il motore Prolog e i database relazionali e il translator è chiamato automaticamente quando serve.

Il codice della libreria è un adattamento del Prolog to SQL translator scritto da Cristoph Draxler e reperibile liberamente all'indirizzo <http://www.j-paine.org/prolog/sql/sql.tar.gz>. Il translator sarà analizzato meglio nella sezione 2.2.

2.1.3 Low-level socket interface to SQL/ODBC databases

Questa libreria fornisce l'interfaccia di basso livello verso MySQL. Utilizza le MySQL C API e l'interfaccia di CIAO Prolog verso il C. I predicati principali sono:³

- `mysql_connect/5`
Si connette al database specificato
- `mysql_query/3`
Esegue una query sul database specificato

³la documentazione di CIAO Prolog non specifica i parametri di questi due predicati.

- `mysql_disconnect/1`
Si disconnette dal database specificato

2.1.4 Altre librerie

Le due restanti librerie contengono predicati non utilizzati direttamente dall'utente che servono per mappare i tipi di dato utilizzati dal translator con i tipi di dato SQL.

2.2 Prolog to SQL translator

In questa sezione sarà analizzato il translator originale, quello scritto da Cristoph Draxler. Lo scopo è capirne il funzionamento, in modo tale da poter essere adattato e infine utilizzato come strumento per mappare il linguaggio SQL su termini e predicati Prolog anche nella nostra libreria `tuProlog`. Alla fine saranno presentati anche alcuni esempi di utilizzo del translator.

2.2.1 Il predicato `translate/3`

`translate/3` è il predicato top-level del translator:

- `translate(+ProjectionTerm, +DatabaseGoal, -SQLQueryTerm)`
`ProjectionTerm` rappresenta il termine sul quale è proiettato il risultato della query al database (come nel caso del primo argomento di `findall/3`); in altre parole indica il pattern che devono seguire le tuple risultato della query. `DatabaseGoal` è il goal Prolog che deve essere tradotto in codice SQL. `SQLQueryTerm` è il parametro di uscita del predicato ed è un termine Prolog che rappresenta la query SQL.

ProjectionTerm - Il termine `ProjectionTerm` può avere come argomenti solo variabili e costanti, non termini composti (il

translator non supporta tipi di dato strutturati come ad esempio le date, ma solo tipi semplici come stringhe o numeri).

DatabaseGoal - Il termine **DatabaseGoal** può contenere congiunzioni, disgiunzioni e negazioni di:

- goal atomici che hanno come funtore il termine corrispondente al nome di una tabella del database e come argomenti variabili o termini istanziati (non composti). Se un argomento è una variabile, questa può essere presente in altri goal atomici di **DatabaseGoal**; se invece l'argomento è un termine istanziato, questo deve essere istanziato a una costante del tipo di dato del translator dichiarato per quel particolare argomento. La funzione di mapping tra funtori dei goal atomici e nomi delle tabelle del database e tra tipi degli argomenti dei goal atomici e tipi delle colonne delle tabelle è svolta dai due predicati cui si fa riferimento alla sezione 2.2.3;
- goal di uguaglianza o disuguaglianza, in cui il funtore principale è un operatore tra quelli permessi (=, \=, <, >, =<, >=, @<, @>) e gli argomenti possono essere costanti semplici (interi, reali, stringhe), variabili oppure termini di aggregazione SQL. Questi hanno come funtore un termine corrispondente a una funzione di aggregazione SQL (avg, min, max, sum, count) e come argomenti una variabile che indica l'argomento sul quale la funzione di aggregazione è calcolata e un termine che deve avere tra gli argomenti la variabile stessa;
- chiamate al predicato **is/2**. Se il primo argomento è una variabile, questa sarà istanziata al valore derivante dalla valutazione del secondo argomento; se è una costante sarà invece semplicemente testata la condizione di uguaglianza. Il secondo argomento può invece essere, come sopra, una costante, una variabile oppure un termine di aggregazione SQL.

In aggiunta per le variabili si può usare il quantificatore esistenziale $\exists/2$ (il predicato $\exists/2$ è vero se il primo argomento è una variabile ed esiste almeno un valore assegnabile a tale variabile in modo che il predicato presente come secondo argomento sia vero). Per un esempio di utilizzo del quantificatore esistenziale, fare riferimento all'esempio 5 della sezione 2.2.4.

2.2.2 Il predicato `printqueries/1`

Attraverso il predicato `translate/3` si ottiene come detto un termine Prolog che rappresenta la query SQL (`SQLQueryTerm`). Per ottenere il codice SQL corrispondente basta passare `SQLQueryTerm` come argomento del predicato `printqueries/1`:

- `printqueries(+Code)`
Stampa sullo standard output il codice SQL corrispondente a `Code`.

Naturalmente la stampa sullo standard output del codice SQL non è molto utile; servirebbe invece ad esempio avere il codice in una variabile per essere usato in altri predicati (ad esempio in quelli che si interfacciano direttamente al database). È questa una delle modifiche che andranno fatte al translator per essere utilizzato nella libreria.

2.2.3 Mapping di tabelle e colonne

Per costruire le query il translator deve conoscere la corrispondenza tra i predicati Prolog e le tabelle nel database. A questo scopo per il corretto funzionamento del translator vanno definite due relazioni, `relation/3` e `attribute/4`. Queste possono essere aggiunte dall'utente alla base di conoscenza del motore attraverso i predicati `assert/1` o `consult/1`.

`relation/3` mappa i nomi dei predicati Prolog nei nomi delle tabelle del database. `attribute/4` mappa le posizioni degli

argomenti di un predicato Prolog nelle colonne della tabella corrispondente nel database. I tipi degli argomenti del predicato Prolog devono essere specificati, e questa informazione è usata durante la traduzione per un controllo di consistenza dei tipi e per la formattazione dell'output del codice. A questo provvede un sistema di tipi minimale presente nel translator.

- `relation(+PredName, +Arity, +TableName)`
PredName è il nome Prolog scelto per una tabella del database. Arity è il numero di argomenti del predicato. TableName è il nome della tabella nel DBMS.
Template: `relation(+atom, +integer, +string)`
- `attribute(+ANumber, +TblName, +AName, +AType)`
ANumber è la posizione dell'attributo nel predicato Prolog, TblName è il nome della tabella nel DBMS. AName è il nome della colonna corrispondente nella tabella. AType è il tipo di dato dell'attributo⁴.
Template: `attribute(+integer, +string, +string, +atom)`

2.2.4 Esempi di utilizzo del translator

Il database utilizzato negli esempi è così definito:

```
relation(flight, 4, 'FLIGHT').
relation(plane, 2, 'PLANE').
attribute(1, 'FLIGHT', 'FLIGHT_NO', string).
attribute(2, 'FLIGHT', 'DEPARTURE', string).
attribute(3, 'FLIGHT', 'DESTINATION', string).
attribute(4, 'FLIGHT', 'PLANE_TYPE', string).
```

⁴AType è il tipo di dato del translator, non il tipo di dato SQL. I tipi di dato supportati dal translator sono: `string`, `integer`, `real`, `natural`, `number`.

```
attribute(1, 'PLANE', 'TYPE', string).
attribute(2, 'PLANE', 'SEATS', integer).
```

La combinazione delle relazioni `relation/3` e `attribute/4` consente di mappare dei termini Prolog in tabelle e colonne del database relazionale. In questo caso nel database esistono due tabelle:

- la tabella `FLIGHT` ha quattro colonne di tipo `string` (`FLIGHT_NO`, `DEPARTURE`, `DESTINATION`, `PLANE_TYPE`);
- la tabella `PLANE` ha due colonne (`TYPE` di tipo `string` e `SEATS` di tipo `integer`).

Esempio 1 – Goal positivi

Goal:

```
translate(flight(No, Dep, Dest, Type),
          flight(No, Dep, Dest, Type),
          Code),
printqueries(Code).
```

Chiedo le informazioni riguardanti tutti i voli. Ogni tupla risultato avrà come funtore principale `flight`.

Output:

```
SELECT rel1.FLIGHT_NO, rel1.DEPARTURE,
       rel1.DESTINATION, rel1.PLANE_TYPE
FROM   FLIGHT rel1
```

Esempio 2 – Join e clausola WHERE

Goal:

```
translate(capacity(No, Dep, Dest, Type,
                  Seats),
          (flight(No, Dep, Dest, Type),
           plane(Type, Seats), Type='b-737',
           Seats < 100),
          Code),
printqueries(Code).
```

Chiedo numero di volo, partenza, destinazione, tipo e numero di posti dell'aereo riguardanti i voli in cui l'aereo è di tipo 'b-737' e il numero di posti è inferiore a 100. Ogni tupla risultato avrà come funtore principale *capacity*.

Output:

```
SELECT rel1.FLIGHT_NO, rel1.DEPARTURE,
       rel1.DESTINATION, rel1.PLANE_TYPE,
       rel2.SEATS
FROM   FLIGHT rel1, PLANE rel2
WHERE  rel2.TYPE = rel1.PLANE_TYPE
AND    rel1.PLANE_TYPE = 'b-737'
AND    rel2.SEATS < 100
```

Esempio 3 – Goal negativi

Goal:

```
translate(no_planes(No, Dep, Dest, Type),
          (flight(No, Dep, Dest, Type),
           not plane(Type, Seats)),
          Code),
printqueries(Code).
```

Chiedo le informazioni riguardanti i voli il cui tipo di aereo non è presente nella tabella degli aerei. Ogni tupla risultato avrà come principale *no_planes*.

Output:

```
SELECT rel1.FLIGHT_NO, rel1.DEPARTURE,
       rel1.DESTINATION, rel1.PLANE_TYPE
FROM   FLIGHT rel1
WHERE  NOT EXISTS (SELECT *
                   FROM   PLANE rel2
                   WHERE  rel2.TYPE =
                        rel1.PLANE_TYPE)
```

Esempio 4 – Funzioni di aggregazione SQL

Goal:

```
translate(X,
```

```
        X is avg(S, plane(T, S)),  
        Code),  
printqueries(Code).
```

Chiedo il numero medio di posti degli aerei.

Output:

```
SELECT AVG(rel1.SEATS)  
FROM    PLANE rel1
```

Esempio 5 – Quantificatore esistenziale

Goal:

```
translate(munich_flight(munich, Dest, Type),  
        FNo^flight(FNo, munich, Dest,  
        Type),  
        Code),  
printqueries(Code).
```

Chiedo le informazioni riguardanti i voli che partono da Monaco. Non m'interessa però il numero del volo. Ogni tupla risultato avrà come funtore principale `munich_flight`.

Output:

```
SELECT 'munich', rel1.DESTINATION,  
        rel1.PLANE_TYPE  
FROM    FLIGHT rel1  
WHERE   rel1.DEPARTURE = 'munich'
```

Capitolo 3

Realizzazione della libreria

Il capitolo ripercorre le fasi della realizzazione della libreria: progetto, sviluppo e collaudo.

3.1 Sviluppo di nuove librerie tuProlog

Le librerie tuProlog sono basate sulla riflessione e possono essere scritte sia in Prolog sia in Java. Per utilizzare lato Prolog i predicati presenti in una libreria scritta in Java tutto quello che serve è la presenza del `.class` della libreria nella corretta posizione del file system.

Una libreria scritta in Java deve estendere la classe astratta `alice.tuprolog.Library` e definisce nuovi *predicati*, *funtori valutabili* e *direttive* sotto forma di metodi, la cui firma segue una semplice convenzione. In particolare i nuovi predicati devono aderire alla signature:

```
public boolean <pred_name>_<N>(T1 arg1, T2  
arg2, ..., Tn argN)
```

mentre i funtori valutabili devono seguire la forma:

```
public Term <eval_funct_name>_<N>(T1 arg1, T2  
arg2, ..., Tn argN)
```

e le direttive devono essere fornite attraverso la firma:

```
public void <dir_name>_<N>(T1 arg1, T2 arg2,  
..., Tn argN)
```

dove $T_1, T_2 \dots T_n$ appartengono alla classe `Term` o a classi derivate come `Struct`, `Var`, `Long`, etc., definite nel package `alice.tuprolog` e che costituiscono la controparte Java dei corrispondenti tipi di dato Prolog. I parametri rappresentano gli argomenti attuali passati ai predicati, funtori o direttive. N è il numero di parametri del metodo.

Inoltre una libreria può definire anche un nuovo pezzo di teoria, acquisita dal motore Prolog attraverso una chiamata al metodo `String getTheory()`. Di default il metodo ritorna una stringa (teoria) vuota; le librerie che vogliono aggiungere un nuovo pezzo di teoria devono sovrascrivere opportunamente il metodo. La stringa di ritorno contiene il codice Prolog che definisce il nuovo pezzo di teoria. I predicati forniti sotto forma di codice Prolog possono chiamare i predicati forniti come metodi Java.

Il nome della libreria normalmente coincide con il nome completo della classe che la implementa.

3.2 Progetto

Un buon punto di partenza per il progetto della libreria per l'accesso a database relazionali è la libreria di CIAO Prolog analizzata nel capitolo 2. Ricordiamo che questa è costituita da un insieme di librerie:

- *SQL persistent database interface* è la libreria di più alto livello, in altre parole quella utilizzata direttamente dagli utenti. Fornisce i predicati `dbassertz_fact/1`, `dbretract_fact/1`, `dbretractall_fact/1`, `dbfindall/4`, `sql_get_tables/2`, `sql_table_types/3`;
- *Prolog to SQL translator* si occupa di tradurre le query Prolog in codice SQL. Fornisce il predicato `pl2sqlString/3`;
- *Low-level socket interface to SQL/ODBC databases* fornisce l'interfaccia di basso livello verso MySQL usando le MySQL C API e l'interfaccia di CIAO Prolog verso il C. I predicati

principali sono `mysql_connect/5` e
`mysql_disconnect/1`;

- *Types for the Low-level interface to SQL databases* e *sqltypes* contengono predicati non utilizzati direttamente dall'utente, che servono per mappare i tipi di dato utilizzati dal translator con i tipi di dato SQL.

L'idea è fornire all'utente un insieme di API costituito da predicati simili a quelli presenti nella *SQL persistent database interface*, utilizzando una versione adattata del translator per generare il codice SQL. Per l'interfacciamento verso i database la scelta ottimale è JDBC, che garantisce l'indipendenza dal particolare database relazionale: pertanto saranno supportati tutti i DBMS per i quali esiste in circolazione un driver JDBC, e non solo MySQL come accade nel caso della libreria di CIAO Prolog. È naturalmente necessario il mapping dei tipi di dato utilizzati dal translator con i tipi di dato JDBC. Vediamo ora le principali scelte progettuali della libreria.

3.2.1 Utilizzo di JDBC

JDBC può essere utilizzato lato Prolog (passando attraverso i predicati della `JavaLibrary` che permettono di utilizzare oggetti Java da programmi Prolog) oppure direttamente in Java. Vista la possibilità offerta da `tuProlog` di creare librerie ibride, cioè classi con metodi Java uniti a codice Prolog incapsulato nel metodo `getTheory()`, è più opportuno usare JDBC direttamente da Java.

Serviranno quindi metodi (nella forma di predicati) che utilizzino JDBC per gestire la connessione al database, la disconnessione e le query di SELECT, INSERT e DELETE. Non è invece necessario un metodo che gestisca la query di UPDATE. Infatti, mentre esistono predicati che eseguono ricerche sul database, inseriscono righe in tabelle o le cancellano, non esiste invece nessun predicato che aggiorni tuple. Un simile predicato potrebbe essere fornito, ma per mantenere una simmetria con i predicati di gestione della knowledge-base (esistono `assert` e

`retract`, ma non predicati di aggiornamento) si è scelto di non fornire tale tipo di predicato, preferendo gestire gli update attraverso la combinazione dei predicati di cancellazione e inserimento. Infine dovrà essere fornita una direttiva per impostare il sottoprotocollo della URL di connessione JDBC (la cui sintassi standard è della forma `jdbc: <subprotocol>: <subname>`) e il driver JDBC.

3.2.2 API e translator

Le API fornite all'utente utilizzano il translator per generare il codice SQL e i predicati (metodi) JDBC per accedere alle basi di dati. Poiché sia le API sia il translator contengono predicati che fanno uso del backtracking, è necessario implementare queste due parti della libreria sotto forma di codice Prolog da incapsulare nella stringa ritornata dal metodo `getTheory()`.

3.2.3 Modifiche da apportare al translator

Come anticipato nel capitolo 3, sarebbe utile avere il codice SQL generato dal translator in una variabile invece che essere stampato sullo standard output. Per ottenere questo è necessario aggiungere alla libreria un campo privato di tipo stringa che rappresenta la query SQL e sostituire in `printqueries(Code)` tutte le occorrenze di `write(Term)` con un nuovo predicato `build_query(Term)` che costruisce man mano la query appendendo `Term` alla stringa. Servirà poi un predicato `get_query(SQLQueryString)` per recuperare la stringa finale completa rappresentante il codice SQL. I due predicati sopra citati, operando su un campo della libreria, vanno implementati in Java.

3.2.4 Gestione dei tipi di dato

La funzione di mapping dei tipi di dato utilizzati dal translator con i tipi di dato SQL deve essere trasparente all'utilizzatore, e quindi

sarà svolta da due metodi privati: uno chiamato all'atto dell'inserimento delle tuple nel database, l'altro nel procedimento inverso. I due metodi saranno chiamati dai predicati JDBC.

3.2.5 Campi di istanza e altre utilità

Infine è prevista la presenza di un altro metodo privato `String javaString(Term string)` che ha la funzione di rappresentare un termine Prolog sotto forma di stringa Java utilizzabile per gli scopi della libreria.

Per quanto riguarda invece i campi di istanza, oltre alla stringa rappresentante il codice SQL serviranno altri campi per memorizzare le informazioni necessarie: il driver JDBC, il sottoprotocollo utilizzato, la connessione, gli oggetti `Statement`, i risultati dell'esecuzione delle query al database.

3.2.6 Diagramma UML

Dalla discussione precedente segue il diagramma progettuale della libreria mostrato in figura 3.1.

3.3 Sviluppo

Analizziamo ora in dettaglio l'implementazione della classe `SQLLibrary`.

3.3.1 Campi di istanza

- `private String subprotocol`
Memorizza il sottoprotocollo della URL di connessione JDBC.
- `private String driver`
Memorizza la stringa rappresentante il driver JDBC.
- `protected Connection connection`

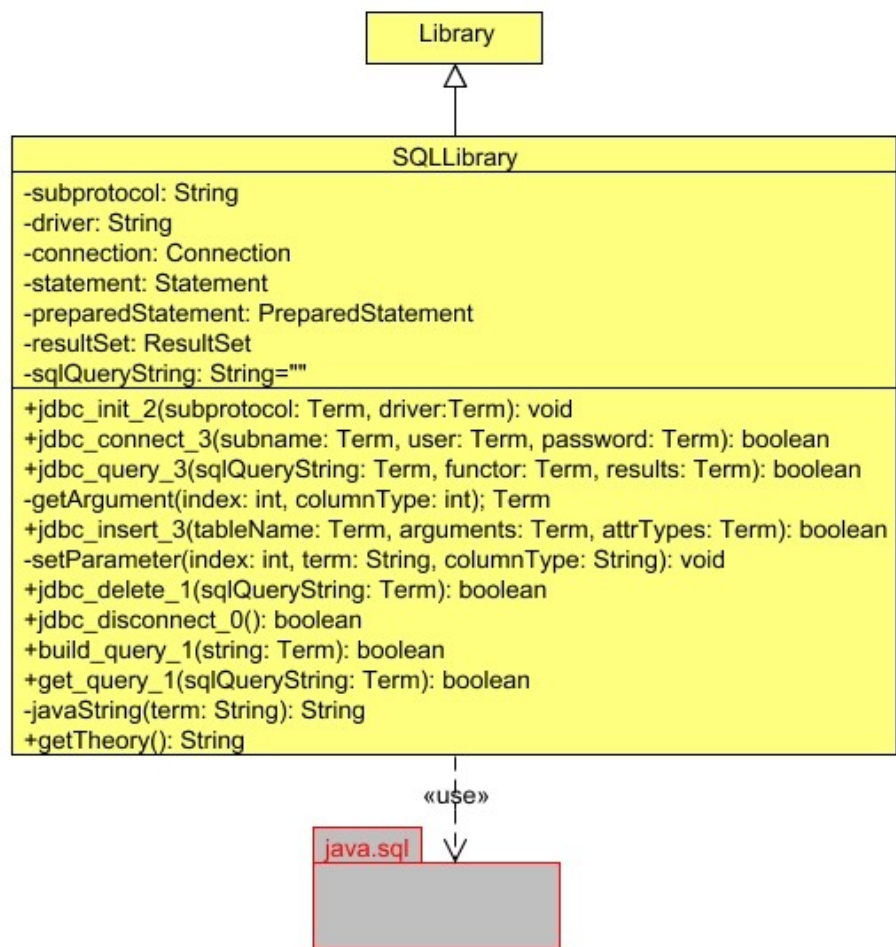


Figura 3.1: Diagramma progettuale della libreria

Memorizza la connessione corrente al database.

- `protected Statement statement`
Memorizza l'oggetto di tipo `Statement` riferito alla connessione.
- `private PreparedStatement preparedStatement`
Memorizza l'oggetto di tipo `PreparedStatement` necessario per eseguire query di parametriche sul database.
- `private ResultSet resultSet`
Memorizza l'oggetto `ResultSet` utilizzato per recuperare i risultati dell'esecuzione di query di `SELECT` sul database.
- `private String sqlQueryString`
Memorizza la stringa rappresentante la query SQL da eseguire sul database. La stringa è inizialmente vuota ed è costruita attraverso chiamate successive del metodo `String build_query_1(Term string)`, che appende `string` a `sqlQueryString`.

3.3.2 Direttive

- `public void jdbc_init_2(Term subprotocol, Term driver)`
Imposta i campi privati `subprotocol` e `driver`.

3.3.3 Predicati

`tuProlog` non prevede la gestione delle eccezioni: ogni eccezione lanciata da un predicato lo fa fallire. Ogni eccezione lanciata da un predicato della libreria causerà pertanto l'immediato fallimento di questo.

- `public boolean jdbc_connect_3(Term subname, Term user, Term password)`
 Cerca di stabilire una connessione al database rappresentato dalla URL JDBC `jdbc:subprotocol:subname`, utilizzando `user` e `password` come credenziali di accesso. Ritorna `true` se la connessione è stata stabilita. Lancia una `ClassNotFoundException` se il driver JDBC non può essere localizzato. Lancia una `SQLException` se avviene un errore di accesso al database (errore nella URL di connessione e/o nelle credenziali di accesso).
- `public boolean jdbc_query_3(Term sqlQueryString, Term functor, Term results)`
 Esegue `sqlQueryString` e i risultati dell'esecuzione della query sono memorizzati nella lista `results`, i cui termini hanno tutti lo stesso funtore `functor`. Al suo interno è utilizzato un oggetto `ResultSetMetaData` per memorizzare i tipi di dato delle colonne risultato dell'esecuzione della query, che non sono noti a priori. Tale oggetto è poi utilizzato per popolare `results` con termini Prolog strutturati i cui argomenti sono di tipi di dato Prolog corrispondenti ai tipi JDBC presenti nell'oggetto `ResultSetMetaData`. Ritorna `true` se l'unificazione di `results` va a buon fine, `false` se `functor` non è un atomo. Lancia una `Exception` se avviene un errore di accesso al database (sintassi SQL errata), se la query è tentata attraverso una connessione al database precedentemente chiusa oppure se almeno una colonna delle tuple risultato è di un tipo di dato SQL corrispondente a un tipo di dato JDBC non supportato dalla libreria.

- `public boolean jdbc_insert_3(Term tableName, Term arguments, Term attrTypes)`

Inserisce la tupla `arguments` nella tabella `tableName`. `attrTypes` contiene i tipi di dato delle colonne della tupla. Tali tipi sono quelli supportati dal translator.

Ritorna `true` se l'inserimento va a buon fine, `false` se `arguments` non è completamente istanziato oppure se `arguments` e `attrTypes` non sono liste della stessa lunghezza.

Lancia una `Exception` se avviene un errore di accesso al database (la tupla da inserire è già presente nel database, non si riescono a convertire i tipi del translator in tipi JDBC, oppure la cardinalità di `arguments` e della tabella `tableName` nel database non è la stessa), se l'inserimento è tentato attraverso una connessione al database precedentemente chiusa, oppure se in `attrTypes` ci sono tipi di dato non supportati dal translator (la libreria non saprebbe in quale tipo di dato JDBC mapparli).

- `public boolean jdbc_delete_1(Term sqlQueryString)`

`sqlQueryString` è una query di `SELECT` generata dal translator. Da questa il metodo costruisce una corrispondente query di `DELETE`, che è eseguita sul database. La modifica della query consiste nel rimpiazzare la sottostringa `"SELECT *"` con la stringa `"DELETE"`, e, attraverso l'utilizzo delle espressioni regolari, nella cancellazione di tutti gli alias delle tabelle (presenti nella query di `SELECT` generata dal translator, ma non permessi nella sintassi della query di `DELETE`). La query di `DELETE` poteva essere costruita anche indipendentemente da quella di `SELECT`; tuttavia, data la somiglianza sintattica delle due query, si è preferito "sfruttare" il translator.

Ritorna `true` se sono state cancellate delle tuple dal database, `false` altrimenti.

Lancia una `SQLException` se avviene un errore di accesso al database (errore nella sintassi SQL) o se la cancellazione è tentata attraverso una connessione al database precedentemente chiusa.

- `public boolean jdbc_disconnect_0()`
Tenta di disconnettersi dal database.
Ritorna `true` se la disconnessione va a buon fine.
Lancia una `SQLException` se avviene un errore di accesso al database (cioè se non esiste nessuna connessione).
- `public boolean build_query_1(Term string)`
Appende `string` a `sqlQueryString`.
Ritorna sempre `true`.
- `public boolean get_query_1(Term sqlQueryString)`
Unifica `sqlQueryString` con l'omonimo campo privato, e riporta tale campo alla condizione iniziale di stringa vuota.
Ritorna `true` se l'unificazione va a buon fine, `false` altrimenti.

3.3.4 Metodi privati

- `private Term getArgument(int index, int columnType)`
Recupera dal campo privato `resultSet` il termine corrispondente alla colonna di indice `index`, utilizzando la "get" corrispondente al tipo di dato JDBC `columnType`.
I tipi di dato JDBC supportati sono: `VARCHAR`, `LONGVARCHAR`, `VARBINARY`, `LONGVARBINARY`,

BINARY, INTEGER, SMALLINT, TINYINT, BIGINT, REAL, FLOAT, DOUBLE.

È chiamato da `jdbc_query_3()`.

Ritorna il termine corrispondente alla colonna di indice `index` e di tipo JDBC corrispondente a `columnType`.

Lancia una `Exception` se `columnType` rappresenta un tipo di dato JDBC non supportato.

- `private void setParameter(int index, String term, String columnType)`

Assegna al parametro di indice `index` del campo privato `preparedStatement` il termine corrispondente a `term`, che è la rappresentazione sotto forma di stringa del termine da assegnare. Per fare ciò utilizza la "set" corrispondente al tipo di dato del translator `columnType`.

È chiamato da `jdbc_insert_3()`.

I tipi di dato supportati dal translator sono: `string`, `integer`, `natural`, `real`, `number`.

Lancia una `Exception` se `columnType` è un tipo di dato non supportato dal translator.

- `private String javaString(Term term)`

Ritorna una rappresentazione di `term` sotto forma di stringa Java. Su `term` viene chiamato prima `getTerm()` e poi `toString()`. Infine si utilizza il metodo statico `removeApices()` della classe `alice.util.Tools` per rimuovere gli apici nel caso `term` sia una stringa Prolog.

È chiamato da quasi tutti i metodi pubblici della libreria.

3.3.5 Il metodo `String getTheory()`

Il metodo `getTheory()` ritorna una stringa contenente il codice Prolog che definisce il nuovo pezzo di teoria della libreria. La stringa consiste in una serie di clausole Prolog separate da un carattere di newline.

La maggior parte delle clausole ritornate dal metodo appartengono al translator. A parte piccoli adattamenti sintattici e definizioni di predicati accessori per far funzionare il translator sulla piattaforma tuProlog, la modifica più importante riguarda i predicati `translate/3` e `printqueries/1`. In precedenza per avere una stampa sullo standard output del codice SQL generato dal translator bisognava eseguire la query:

```
translate(ProjectionTerm, DatabaseGoal,  
SQLQueryTerm), printqueries(SQLQueryTerm).
```

Applicando le modifiche discusse nella sezione 3.2.3, la sola chiamata di `translate/3` permette di ottenere il codice SQL dal terzo parametro del predicato:

```
translate(+ProjectionTerm, +DatabaseGoal, -  
SQLQueryString).
```

Questo è ottenuto inserendo come ultimi due goal di `translate/3` le chiamate a `printqueries/1` e `get_query/1`.

I restanti predicati costituiscono l'insieme delle API messe a disposizione dell'utente: `dbassert/1`, `dbretract/1`, `dbcurrent/1`, `dbretractall/1`, `dbfindall/3`, `get_tables/1`, `table_types/3`. La documentazione e gli esempi di utilizzo di questi predicati, oltre che della direttiva `jdbc_init/2` e dei predicati `jdbc_connect/3` e `jdbc_disconnect/0` (forniti sotto forma di metodi Java) sarà argomento del prossimo capitolo. La loro funzione ricalca comunque quella dei predicati corrispondenti della libreria di CIAO Prolog.

3.4 Collaudo

Oltre alla libreria sono state realizzate tre classi di test allo scopo di verificarne il corretto funzionamento. Come strumento di test si è utilizzato JUnit 4.4, le cui funzionalità si integrano alla perfezione con Eclipse, che è invece lo strumento utilizzato per lo sviluppo della libreria.

I test verificano che i predicati della libreria funzionino correttamente; tuttavia, vista la possibilità di interfacciarsi a diversi database, non è possibile scrivere una classe di test che funzioni ovunque. Anzi, esistendo oggetti (come ad esempio la stringa che rappresenta la URL di connessione JDBC) che sono diversi da DBMS a DBMS, per avere la certezza del funzionamento assoluto della libreria saremmo costretti a implementare centinaia di test, e ciò non rientra nello scopo di questa tesi.

Alla luce di queste considerazioni, l'architettura di test scelta fornisce due classi cablate su due specifici DBMS, ma è anche estendibile, permettendo facilmente a chiunque di scrivere una nuova classe di test per un altro DBMS.

A questo scopo esiste una classe astratta (`SQLLibraryTestCase`), che implementa tutti i test tranne quello della connessione, e due classi concrete che la estendono e che rappresentano il concetto di `TestCase` eseguito rispettivamente su un DB MySQL (`SQLLibraryMySQLTestCase`) e su un DB SQLite (`SQLLibrarySQLiteTestCase`). Per fare ciò aggiungono quattro metodi alla classe astratta. Ogni nuova classe di test dovrà pertanto seguire tale pattern: estendere `SQLLibraryTestCase` e implementare gli stessi quattro metodi. Non è possibile dichiarare nella classe base tutti e quattro i metodi come astratti perché due di questi devono essere statici: la parola “chiave” `static` non è consentita nelle classi astratte, e nelle classi che estendono una classe astratta non è possibile aggiungere modificatori ai metodi. I quattro metodi da implementare sono:

1. `public static void setUpBeforeClass()
throws Exception;`

2. `public static void tearDownAfterClass() throws Exception;`
3. `public void setUp() throws Exception;`
4. `public void testConnection();`

I primi tre metodi sono metodi standard di JUnit 4:

1. `setUpBeforeClass()` è eseguito una volta sola, cioè prima del primo metodo di test della classe;
2. `tearDownAfterClass()` è eseguito una volta sola, cioè dopo l'ultimo metodo di test della classe;
3. `setUp()` è eseguito prima di ogni metodo di test della classe.

Infine il quarto metodo collauda insieme connessione e disconnessione; esegue cioè le chiamate ai due predicati corrispondenti della libreria. In figura 3.2 è mostrata l'architettura di test della libreria.

La sezione 3.5.2 contiene il codice dei metodi della classe `SQLLibraryMySQLTestCase` (l'altra è molto simile). In questo modo chi intende scrivere una nuova classe di test può partire da tale codice, e su questo eseguire le modifiche necessarie per adattarlo al particolare DBMS.

3.4.1 La classe astratta `SQLLibraryTestCase`

La classe astratta `SQLLibraryTestCase` ha tre campi privati:

- un oggetto `Prolog`;
- un oggetto `Connection`;
- un oggetto `Statement`.

Il primo campo rappresenta il motore `tuProlog`, su cui chiamare l'esecuzione dei predicati attraverso il metodo `solve(Term g)`. Gli altri due campi rappresentano invece gli oggetti necessari per connettersi al database e inserirvi e cancellarvi tuple.

La documentazione di JUnit dice che l'ordine dei test non è assicurato: i singoli test devono essere indipendenti l'uno dall'altro. Pertanto in ogni test:

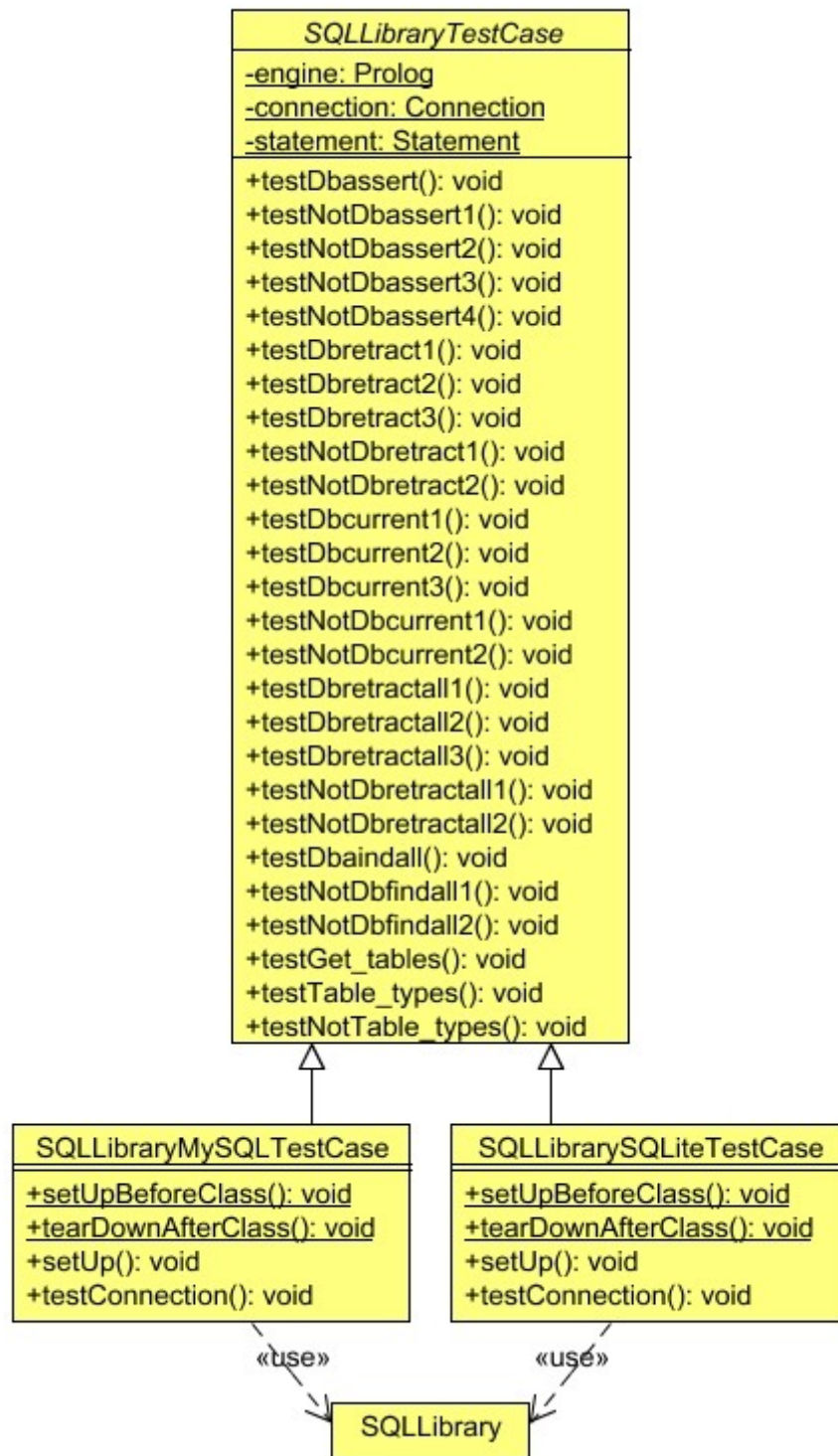


Figura 3.2: Architettura di test della libreria

1. attraverso l'oggetto `Statement` sono inserite nel database le eventuali tuple necessarie (ad esempio tuple da cancellare poi attraverso `dbretract/1`);
2. è eseguito il test (chiamata al metodo `solve()` sull'oggetto `Prolog` e controllo del risultato attraverso i metodi del package `org.junit.Assert`);
3. sempre attraverso l'oggetto `Statement` viene ripulito il database (ad esempio nel caso di tuple aggiunte attraverso `dbassert/1`), in modo che test successivi non falliscano a causa di tuple già presenti.

Secondo questo schema, ogni test “costruisce” quello di cui ha bisogno, e infine cancella tutto. In questo modo ottengo l'indipendenza: l'ordine (casuale) con cui saranno eseguiti i test non è più rilevante.

La classe contiene poi una serie metodi, tutti etichettati con l'annotazione `@test`, che collaudano tutti i predicati della libreria (eccetto la connessione e la disconnessione, che dipendono dal particolare DBMS e che saranno collaudati nelle classi che ereditano dalla classe astratta). Sono eseguiti test sia di correttezza, sia di fallimento dei predicati. Non sono eseguiti test sul translator che, anche se leggermente modificato, si suppone funzioni e sia già stato collaudato in quanto libreria di terze parti.

3.4.2 La classe `SQLLibraryMySQLTestCase`

`SQLLibraryMySQLTestCase` è una classe che estende `SQLLibraryTestCase` e rappresenta il concetto di test della libreria su un database MySQL. Vediamone il codice dei metodi.

```
@BeforeClass
```

```
public static void setUpBeforeClass() throws Exception {
    Class.forName("com.mysql.jdbc.Driver");
    connection =
        DriverManager.getConnection("jdbc:mysql://localhost/tuprologtest", "tuprolog", "tuprolog");
}
```

```

statement = connection.createStatement();
statement.executeUpdate("CREATE TABLE
    `tuprologtest`.`flight` (`FLIGHT_NO`
    VARCHAR(45) NOT NULL, `DEPARTURE` VARCHAR(45)
    NOT NULL, `DESTINATION` VARCHAR(45) NOT NULL,
    `PLANE_TYPE` VARCHAR(45) NOT NULL, PRIMARY KEY
    (`FLIGHT_NO`))");
statement.executeUpdate("CREATE TABLE
    `tuprologtest`.`plane` (`TYPE` VARCHAR(45) NOT
    NULL, `SEATS` INTEGER UNSIGNED NOT NULL,
    PRIMARY KEY (`TYPE`))");
}

```

Il metodo registra il driver JDBC di MySQL e si connette al database usato per il test. Crea poi le tabelle necessarie.

Il database cui ci si connette, lo schema in cui creare le tabelle e l'utente che si connette al database devono già esistere. Non è infatti possibile in generale creare il database, lo schema o l'account direttamente dalla classe perché l'utente che voglia eseguire tali operazioni potrebbe non averne i privilegi necessari.

Nel caso di SQLite questo limite invece non c'è: ci si può connettere anonimamente al database, e la tabella non è creata in uno schema, ma direttamente in memoria. Il test su un database SQLite è pertanto totalmente auto contenuto.

@AfterClass

```

public static void tearDownAfterClass() throws Exception
{
    statement.executeUpdate("DROP TABLE `flight`");
    statement.executeUpdate("DROP TABLE `plane`");
    statement.close();
    connection.close();
}

```

Il metodo cancella le tabelle utilizzate durante il test e chiude la connessione.

```

@Before
public void setUp() throws Exception {
    engine = new Prolog();
    SQLLibrary lib = new SQLLibrary();
    Field field =
        Class.forName("alice.tuprolog.lib.SQLLibrary").
            getDeclaredField("connection");
    field.setAccessible(true);
    field.set(lib, connection);
    field =
        Class.forName("alice.tuprolog.lib.SQLLibrary").
            getDeclaredField("statement");
    field.setAccessible(true);
    field.set(lib, statement);
    engine.loadLibrary(lib);
    engine.addTheory(new Theory(":-
        jdbc_init(sqlite, 'org.sqlite.JDBC')."));
    engine.addTheory(new
        Theory("relation(flight,4, 'FLIGHT')."));
    engine.addTheory(new
        Theory("relation(plane,2, 'PLANE')."));
    engine.addTheory(new
        Theory("attribute(1, 'FLIGHT', 'FLIGHT_NO', string
        )."));
    engine.addTheory(new
        Theory("attribute(2, 'FLIGHT', 'DEPARTURE', string
        )."));
    engine.addTheory(new
        Theory("attribute(3, 'FLIGHT', 'DESTINATION', string
        )."));
    engine.addTheory(new
        Theory("attribute(4, 'FLIGHT', 'PLANE_TYPE', string
        )."));
    engine.addTheory(new
        Theory("attribute(1, 'PLANE', 'TYPE', string)."));
}

```

```

        engine.addTheory(new
            Theory("attribute(2, 'PLANE', 'SEATS', integer).")
        );
    }

```

Il metodo crea un nuovo motore Prolog e un'istanza della libreria da collaudare. Poi, usando la riflessione, rende accessibili i campi privati della libreria `connection` e `statement` e li imposta con i valori dei rispettivi campi ereditati dalla classe superiore. Infine carica nel motore l'istanza della libreria e aggiunge al motore le relazioni necessarie per far funzionare il translator.

Il caricamento di una specifica istanza della libreria è fondamentale per evitare spiacevoli dipendenze. Quasi tutti i predicati della libreria necessitano infatti di una connessione attiva, e questa connessione (rappresentata dall'oggetto `connection`) è creata dal predicato `jdbc_connect/1`. Tuttavia non si può, in un predicato da collaudare, utilizzare un altro predicato che potrebbe però ancora non essere stato collaudato, visto che l'ordine dei test è casuale. Invece, attraverso il meccanismo mostrato nel codice, gli oggetti `connection` e `statement` della libreria sono impostati a valori "sicuri", ottenuti cioè utilizzando metodi contenuti nella libreria standard di Java (nel caso specifico quelli di JDBC).

```

@Test
public void testConnection() {
    Struct subname = new
        Struct("//localhost/tuprologtest");
    Struct user = new Struct("tuprolog");
    Struct password = new Struct("tuprolog");
    SolveInfo goal = engine.solve(new
        Struct("jdbc_connect", subname, user,
            password));
    assertTrue(goal.isSuccess());
    goal = engine.solve(new Struct("jdbc_disconnect"));
    assertTrue(goal.isSuccess());
}

```

Il metodo collauda allo stesso tempo connessione e disconnessione. La connessione creata in questo metodo sovrascriverà nella libreria quella impostata nel metodo `setUp()`, ma la modifica non si ripercuoterà sugli altri metodi di test, che utilizzeranno comunque la connessione creata dal metodo `setUpBeforeClass()`. Infatti ogni metodo di test opera su una differente istanza della libreria (creata nel `setUp()`).

Capitolo 4

Manuale di utilizzo

4.1 Operazioni preliminari

Prima di poter usufruire dei predicati messi a disposizione dalla libreria, l'utente deve eseguire alcune operazioni preliminari.

4.1.1 Caricamento della libreria

La prima operazione da eseguire è naturalmente il caricamento della libreria (che deve essere presente nel CLASSPATH) nel motore tuProlog. Questo può avvenire attraverso la direttiva `load_library/1`:

```
:- load_library('alice.tuprolog.lib.SQLLibrary').
```

Un altro modo per caricare la libreria è sfruttare l'IDE tuProlog. La finestra principale dell'IDE è mostrata in figura 4.1. Cliccando sul quinto pulsante presente nella toolbar (Open Library Manager) si ottiene la finestra di dialogo mostrata in figura 4.2.

L'utente deve digitare `alice.tuprolog.lib.SQLLibrary` nel campo di testo situato in cima alla finestra di dialogo, e poi premere il pulsante **Add**. La libreria aggiunta è inizialmente visualizzata nello stato **Unloaded**. L'utente deve modificarne lo stato in **Loaded**, e infine premere il pulsante **OK**.

Infine è da segnalare per i programmatori Java la possibilità di caricare la libreria chiamando il metodo `loadLibrary()` della classe `alice.tuprolog.Prolog`.

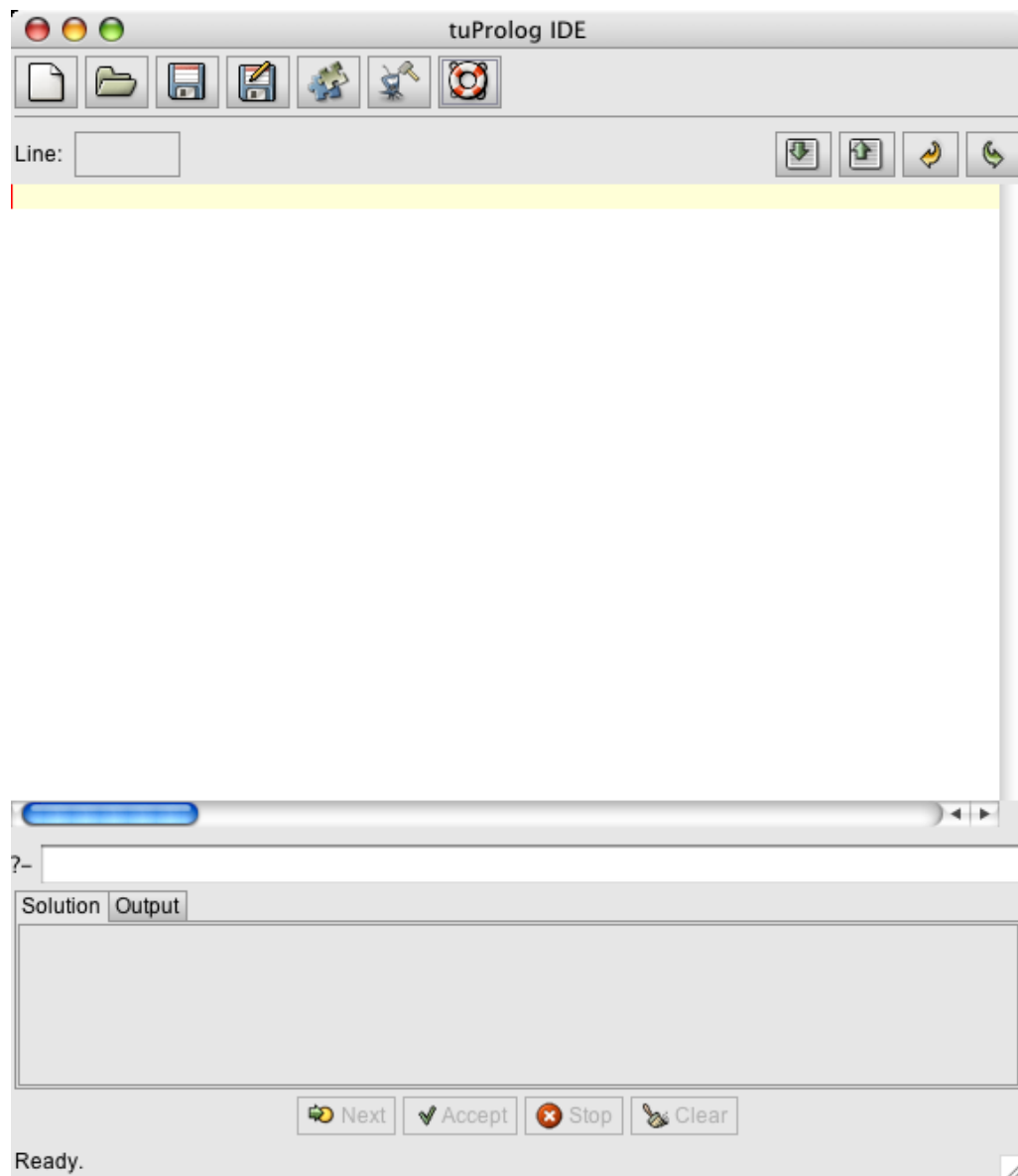


Figura 4.1: L'IDE di tuProlog

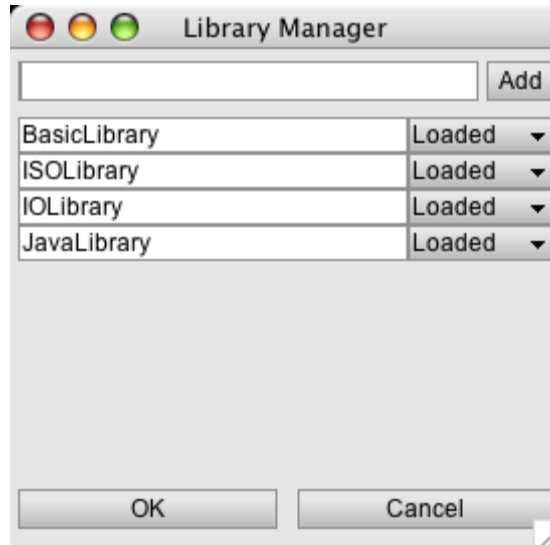


Figura 4.2: La finestra di dialogo Open Library Manager

4.1.2 Mapping di tabelle e colonne del database

La libreria utilizza un translator per trasformare in codice SQL le query espresse dall'utente in linguaggio Prolog. Perché la traduzione vada a buon fine è però necessario che l'utente fornisca al translator le informazioni necessarie per il mapping di tabelle e colonne del database. Queste informazioni consistono di una serie di fatti, espressi attraverso le relazioni `relation/3` e `attribute/4`.⁵

Tali fatti devono essere aggiunti alla base di conoscenza del motore, e l'utente ha diverse possibilità per fare questo:

- aggiungere ogni fatto attraverso il predicato `assert/1`;
- racchiudere tutti i fatti in un file di testo e consultarlo utilizzando `consult/1`;
- sfruttare l'IDE di tuProlog. In questo caso i fatti devono essere inseriti nell'area di testo posta al centro della finestra principale dell'IDE;

⁵Per la documentazione e l'utilizzo delle due relazioni si vedano le sezioni 2.2.3 e 2.2.4.

- chiamare il metodo `addTheory()` della classe `alice.tuprolog.Prolog`, nel caso di utilizzo del motore `tuProlog` da programmi Java.

Naturalmente le tabelle così mappate devono esistere in qualche database.

4.1.3 Driver JDBC

Le ultime operazioni riguardano il driver JDBC specifico per il particolare DBMS cui ci si vuole interfacciare. È necessario procurarsi il driver e assicurarsi che la classe corrispondente sia presente nel `CLASSPATH`. Se così non è, tutti i predicati che hanno bisogno di accedere al database falliranno.

Fatto ciò, bisogna impostare il sottoprotocollo JDBC che si intende utilizzare e registrare il driver JDBC. Il sottoprotocollo JDBC è la seconda parte della URL di connessione JDBC, la cui sintassi standard è `jdbc:<subprotocol>:<subname>`. Le due operazioni sono eseguite attraverso la direttiva `jdbc_init/2`.

- `:- jdbc_init/2`
`jdbc_init(Subprotocol, Driver)` memorizza nel motore il sottoprotocollo JDBC che si intende utilizzare (`Subprotocol`) e registra il driver JDBC (`Driver`). `Subprotocol` è il nome del driver. `Driver` è una stringa Prolog indicante il nome completo della classe del driver JDBC.
 Per il nome esatto del driver (`Subprotocol`) si consiglia di consultare il manuale dello specifico DBMS da utilizzare.
Template: `jdbc_init(+atom, +string)`

L'utente deve inserire la direttiva nell'area di testo posta al centro della finestra principale dell'IDE. Utilizzando più volte la direttiva nell'area di testo è possibile registrare più driver JDBC; tuttavia il motore è in grado di memorizzare un solo sottoprotocollo. Se

nell'area di testo sono presenti più direttive, il motore prende in considerazione soltanto l'ultima. È tuttavia consigliabile mantenere una sola direttiva, ed eventualmente modificarla se a un certo punto si vuole cambiare DBMS.

4.2 Predicati

Segue ora una descrizione dei predicati messi a disposizione dalla libreria. Alcuni di questi nella loro implementazione fanno uso di predicati presenti in altre librerie tuProlog. Queste librerie devono essere pertanto caricate nel motore per assicurare un corretto funzionamento della `SQLLibrary`. Le dipendenze di questa libreria sono `BasicLibrary` e `ISOLibrary`.

- `jdbc_connect/3`
`jdbc_connect(Subname, User, Password)` è vero se il motore riesce a stabilire, utilizzando `User` e `Password` come credenziali di accesso, una connessione al database rappresentato dalla URL JDBC `jdbc:<subprotocol>:Subname`. `subprotocol` deve essere stato impostato dalla direttiva `jdbc_init/2`. `Subname` è un modo per identificare la base di dati. La sua sintassi può variare a seconda del sottoprotocollo, ma deve comunque fornire abbastanza informazioni per localizzare la sorgente dati. Normalmente `Subname` segue la sintassi `//hostname:port/subsubname`, dove `subsubname` è il nome dello schema all'interno del database. Si consiglia comunque di consultare il manuale dello specifico DBMS da utilizzare.
Template: `jdbc_connect(+string, +string, +string)`
- `jdbc_disconnect/0`

`jdbc_disconnect` è vero se il motore riesce a disconnettersi dal database. Fallisce se non esiste nessuna connessione.

Template: `jdbc_disconnect`

- `dbassert/1`

`dbassert(Fact)` è vero se l'istanza corrente di `Fact`, interpretata come una tupla di una relazione nel database, è aggiunta alla tabella corrispondente al funtore principale di `Fact`. La corrispondenza tra nome della tabella e funtore è ricavata dalla relazione `relation/3`. Il predicato fallisce se `Fact` non è completamente istanziato o è già presente nel database; fallisce inoltre se non si riescono a mappare i tipi di dato degli argomenti di `Fact` (dichiarati mediante la relazione `attribute/4`) con i rispettivi tipi di dato JDBC delle tabelle del database.

Template: `dbassert(@compound)`

- `dbretract/1`

`dbretract(Fact)` è vero se nel database esiste un fatto che unifica con `Fact`, e lo cancella. Attraverso il backtracking è possibile cancellare dal database tutti i fatti che unificano con `Fact`. Usa `dbcurrent/1` per verificare l'esistenza del fatto. `Fact` può essere completamente istanziato, o in parte, o per niente.

Template: `dbretract(+compound)`

- `dbcurrent/1`

`dbcurrent(Fact)` è vero se nel database corrente esiste un fatto che unifica con `Fact`. Attraverso il backtracking recupera dal database tutti i fatti che unificano con `Fact`. Usa `dbfindall/3` per eseguire la query sul database. `Fact` può essere completamente istanziato, o in parte, o per niente.

Template: `dbcurrent(+compound)`

- **dbretractall/1**
dbretractall(Fact) è vero se nel database esiste almeno un fatto che unifica con **Fact** (che può essere completamente istanziato, o in parte, o per niente), e cancella dal database tutti i fatti che unificano con **Fact**.
Template: **dbretractall(+compound)**
- **dbfindall/3**
dbfindall(Pattern, ComplexGoal, Results) è vero se **Results** contiene tutte le tuple che rappresentano la risposta dal database alla query Prolog **ComplexGoal**, proiettate su **Pattern** (ogni tupla presente in **Results** deve seguire il pattern indicato da **Pattern**). Usa il translator per generare la query SQL partendo da **ComplexGoal**. Il predicato è simile a **findall/3**, soltanto che **ComplexGoal** è eseguito su un database relazionale. Il predicato fallisce se almeno un argomento delle tuple risultato è di un tipo di dato SQL corrispondente a un tipo di dato JDBC non supportato dalla libreria. I tipi di dato JDBC supportati dalla libreria sono: **VARCHAR**, **LONGVARCHAR**, **VARBINARY**, **LONGVARBINARY**, **BINARY**, **INTEGER**, **SMALLINT**, **TINYINT**, **BIGINT**, **REAL**, **FLOAT**, **DOUBLE**.
Template: **dbfindall(+term, +callable_term, -list)**
- **get_tables/1**
get_tables(Tables) è vero e **Tables** è la lista degli atomi Prolog corrispondenti alle tabelle nel database corrente. Il mapping avviene attraverso la relazione **relation/3**.
Template: **get_tables(-list)**
- **table_types/3**

`table_types(Table, AttrNames, AttrTypes)` è vero se `AttrNames` e `AttrTypes` sono le liste contenenti i nomi e i tipi di dato del translator delle colonne della tabella `Table`. `Table` è il nome Prolog di una tabella del database corrente, e deve essere presente nel programma come argomento di una relazione `relation/3`.

Template: `table_types(+atom, -list, -list)`

4.3 Esempi di utilizzo dei predicati

Vediamo infine alcuni esempi di utilizzo dei predicati della libreria. Il database utilizzato negli esempi seguenti è lo stesso di quello degli esempi della sezione 2.2.4.

Ipotizziamo che sulla nostra macchina giri un server MySQL, e che esista uno schema di nome `tuprologtest` e un account con username `tuprolog` e password `tuprolog`. Ipotizziamo inoltre che il driver JDBC per MySQL sia incluso nel CLASSPATH. Posso allora impostare la direttiva `jdbc_init/2`:

```
:- jdbc_init(mysql, 'com.mysql.jdbc.Driver').
```

Adesso sono in grado di connettermi al database:

```
jdbc_connect('//localhost/tuprologtest',  
'tuprolog', 'tuprolog').
```

Terminata la sessione, posso disconnettermi dal database:

```
jdbc_disconnect.
```

4.3.1 dbassert/1

Esempio 1

```
dbassert(flight('1', 'Munich', 'Rome',  
'b-737')).
```

Se la tupla ('1', 'Munich', 'Rome', 'b-737') non è presente nella tabella FLIGHT, la aggiunge e il predicato è vero.

Esempio 2

```
dbassert(city('1', 'Munich', 'Rome',  
'b-737')).
```

Fallisce perché la tabella `city` non esiste nel database.

Esempio 3

```
dbassert(plane('c-919', '300')).
```

Fallisce perché il secondo argomento di `plane` deve essere un intero.

Esempio 4

```
dbassert(flight(No, 'Munich', 'Rome',  
'b-737')).
```

Fallisce perché il primo argomento di `flight` non è istanziato.

4.3.2 dbretract/1

Esempio 1

```
dbretract(flight(No, Dep, Dest, Type)).
```

È vero se nella tabella FLIGHT sono presenti tuple, e le rimuove attraverso il backtracking. Ogni volta le variabili sono istanziate ai valori della tupla che unifica con il fatto.

Esempio 2

```
dbretract(flight(No, Dep, 'Rome', 'b-737')).
```

È vero se nella tabella FLIGHT sono presenti tuple per le quali `DESTINATION = 'Rome'` e `PLANE_TYPE = 'b-737'`, e le rimuove attraverso il backtracking. Ogni volta le variabili sono istanziate ai valori della tupla che unifica con il fatto.

Esempio 3

```
dbretract(flight('1', 'Munich', 'Rome',  
'b-737')).
```

È vero se nella tabella FLIGHT è presente la tupla ('1', 'Munich', 'Rome', 'b-737'), e la rimuove.

Esempio 4

```
dbretract(city(No, Dep, Dest, Type)).
```

Fallisce perché la tabella city non esiste nel database.

4.3.3 dbcurrent/1

Esempio 1

```
dbcurrent(flight(No, Dep, Dest, Type)).
```

È vero se nella tabella FLIGHT sono presenti tuple, e le recupera attraverso il backtracking. Ogni volta le variabili sono istanziate ai valori della tupla che unifica con il fatto.

Esempio 2

```
dbcurrent(flight(No, Dep, 'Rome', 'b-737')).
```

È vero se nella tabella FLIGHT sono presenti tuple per le quali DESTINATION = 'Rome' e PLANE_TYPE = 'b-737', e le recupera attraverso il backtracking. Ogni volta le variabili sono istanziate ai valori della tupla che unifica con il fatto.

Esempio 3

```
dbcurrent(flight('1', 'Munich', 'Rome',  
'b-737')).
```

È vero se nella tabella FLIGHT è presente la tupla ('1', 'Munich', 'Rome', 'b-737').

Esempio 4

```
dbcurrent(city(No, Dep, Dest, Type)).
```

Fallisce perché la tabella city non esiste nel database.

4.3.4 dbretractall/1

Esempio 1

`dbretractall(flight(No, Dep, Dest, Type)).`

È vero se nella tabella FLIGHT sono presenti tuple, e le rimuove.

Esempio 2

`dbretractall(flight(No, Dep, 'Rome', 'b-737')).`

È vero se nella tabella FLIGHT sono presenti tuple per le quali `DESTINATION = 'Rome'` e `PLANE_TYPE = 'b-737'`, e le rimuove.

Esempio 3

`dbretractall(flight('1', 'Munich', 'Rome', 'b-737')).`

È vero se nella tabella FLIGHT è presente la tupla `('1', 'Munich', 'Rome', 'b-737')`, e la rimuove.

Esempio 4

`dbretractall(city(No, Dep, Dest, Type)).`

Fallisce perché la tabella `city` non esiste nel database.

4.3.5 dbfindall/3

Esempio 1

```
dbfindall(capacity(No, Dep, Dest, Type,
                    Seats),
          (flight(No, Dep, Dest, Type),
           plane(Type, Seats),
           Type = 'b-737'),
          Results).
```

Esegue la query Prolog:⁶

```
flight(No, Dep, Dest, Type),  
plane(Type, Seats), Type = 'b-737'
```

Results è la lista dei risultati dell'esecuzione della query sul database: sarà una lista vuota oppure una lista di termini che seguono il pattern `capacity(No, Dep, Dest, Type, Seats)`.

Esempio 2

```
dbfindall(capacity(Number, Dep, Dest, Type,  
                    Seats),  
          (flight(No, Dep, Dest, Type),  
           plane(Type, Seats),  
           Type = 'b-737'),  
          Results).
```

Fallisce perché la variabile `Number` del primo argomento non è presente nel secondo argomento.

Esempio 3

```
dbfindall(capacity(No, Dep, Dest, Type,  
                    Seats),  
          (flight(No, Dep, Dest, Type),  
           plane(PlaneType, Seats),  
           Type = 'b-737'),  
          Results).
```

Fallisce perché nessuna variabile della query `plane(PlaneType, Seats)` è collegata ad altre variabili del secondo argomento (e quindi non è possibile fare il join tra le tabelle FLIGHT e PLANE).

⁶I primi due argomenti di `dbfindall/3` sono passati così come sono al translator. Per il significato della query, per la sintassi dei termini e per altri esempi di utilizzo del translator, fare riferimento alla sezione 2.2.

4.3.6 get_tables/1

Esempio 1

```
get_tables(Tables).
```

Il risultato è:

```
Tables = [flight,plane].
```

4.3.7 table_types/3

Esempio 1

```
table_types(flight, AttrNames, attrTypes).
```

Il risultato è:

```
AttrNames = [ 'FLIGHT_NO', 'DEPARTURE',  
              'DESTINATION', 'PLANE_TYPE' ]
```

```
AttrTypes = [string,string,string,string].
```

Esempio 2

```
table_types(city, AttrName, attrTypes).
```

Fallisce perché la tabella `city` non esiste nel database.

Conclusioni

Lo scopo principale della tesi era realizzare una libreria che consentisse di accedere, in maniera trasparente per l'utente, a database relazionali da programmi tuProlog. Infatti, prima della realizzazione della libreria tale accesso era comunque possibile utilizzando i predicati della `JavaLibrary`, ma l'utente doveva conoscere JDBC ed era costretto a utilizzare oggetti Java direttamente da programmi Prolog.

Per giungere a questo obiettivo è stato necessario anzitutto acquisire una buona conoscenza del linguaggio Prolog e dell'architettura tuProlog. La seconda fase è stata l'analisi delle scelte di modello e interfaccia adottate in altri motori Prolog, ed è risultato che solo CIAO Prolog (tra i motori più importanti) mette a disposizione dei predicati per interfacciarsi verso le basi di dati. È stato poi necessario comprendere il funzionamento del Prolog to SQL translator, uno strumento già esistente che permette di tradurre in linguaggio SQL query espresse in linguaggio Prolog. L'ultima fase del lavoro è stata l'implementazione della libreria, in cui si sono dovute effettuare alcune modifiche di dettaglio sul translator per poterlo utilizzare nel contesto tuProlog.

Tutte le funzionalità della libreria sono state collaudate attraverso una serie di test, eseguiti in particolare su DB MySQL e SQLite. Nel caso di MySQL tutti i metodi di test vanno a buon fine, nell'altro caso un metodo di test (`testDbretractall1()`) fallisce per problemi di JDBC nei confronti di SQLite. È presente poi un problema riguardo all'utilizzo delle funzioni aggregate nelle query al database, anche questo però dovuto a terze parti (nel caso specifico a un problema di implementazione di `findall/3` in tuProlog).

Infine si è ritenuto utile mettere a disposizione un manuale di utilizzo della libreria, comprendente anche numerosi esempi di utilizzo dei predicati.

Bibliografia

Database e modello relazionale

- http://it.wikipedia.org/wiki/Modello_relazionale
- <http://www-db.deis.unibo.it/courses/SIL-A/PDF/Relazionale.pdf>

Prolog e programmazione logica

- Bratko, I. (1986) *Prolog programming for artificial intelligence*. Addison-Wesley
- <http://en.wikipedia.org/wiki/Prolog>
- http://en.wikipedia.org/wiki/Logic_programming

tuProlog

- <http://www.alice.unibo.it/tuProlog/>

CIAO Prolog Reference Manual

- <http://www.clip.dia.fi.upm.es/Software/Ciao/ciao.pdf>

Prolog to SQL translator

- <http://www.j-paine.org/prolog/sql/sql.tar.gz>

Java SE 6 Documentation

- <http://java.sun.com/javase/6/docs/>