UNIVERSITÀ DEGLI STUDI DI BOLOGNA

FACOLTÀ DI INGEGNERIA

Corso di Laurea Magistrale in Ingegneria Informatica
Fondamenti di Intelligenza Artificiale M
Prof. Paola Mello, Ing. Federico Chesani, Ing. Marco Montali

Progetto di Fondamenti di Intelligenza Artificiale M

"Realizzazione di una gioco basato su regole comportamentali"

Realizzato da:
Enrico Baioni
Andrea Grandi
Simone Tallevi Diotallevi

ANNO ACCADEMICO 2009/2010

Indice

Introduzione

L'attività progettuale ha lo scopo di realizzare un gioco basato su regole comportamentali. Tali regole non devono essere cablate all'interno del gioco stesso ma possono essere inserite di volta in volta dall'utente, il quale può quindi personalizzarle a proprio piacimento.

Più in dettaglio, il gioco si svolge all'interno di una scacchiera costituita da celle aventi un determinato colore e contenenti un certo oggetto. L'obiettivo del gioco è cercare di spostarsi all'interno di tale scacchiera rispettando le regole immesse e cercando di raggiungere una cella obiettivo, anch'essa specificata dall'utente. Le regole inserite dall'utente possono ad esempio costringere o vietare il passaggio su altre celle nel caso in cui si verifichino determinate condizioni, oppure possono limitare le possibilità di spostamento del giocatore dal momento che inizialmente non è posto nessun vincolo.
Anche la scacchiera su cui si svolge il gioco deve poter essere personalizzata dall'utente, il quale può specificare colore e contenuto di ciascuna cella.

Il programma deve prevedere poi due modalità distinte di funzionamento, in particolare una di verifica e una di generazione. Nella prima l'utente cerca di trovare un percorso valido, cioè che rispetti le regole comportamentali inserite: in questo caso il compito del programma sarà quello di comunicare se il percorso inserito è corretto o meno. La seconda modalità, più interessante dal punto di vista dell'Intelligenza Artificiale, coinvolge principalmente il computer il quale cerca un percorso corretto in modo autonomo.

Per implementare il controllo del comportamento dell'utente e la generazione del percorso da parte del computer è stato quindi sfruttato il framework SCIFF, tramite il quale è possibile esprimere in modo dichiarativo il comportamento di un'entità. L'utente può quindi immettere le regole nel sistema direttamente nel linguaggio di SCIFF oppure in modo semplificato attraverso dei predicati Prolog che saranno opportunamente gestiti per realizzare un set di comportamenti predefiniti.

Per realizzare l'interfaccia grafica del gioco si utilizzerà invece Java, appoggiandosi alla libreria JPL. Quest'ultima viene fornita insieme a Swi-Prolog e consente di invocare comandi Prolog all'interno di un programma Java.

Nei capitoli 1 e 2 di questa relazione si parlerà quindi del framework SCIFF, della libreria JPL e di come sono stati sfruttati all'interno del progetto. Il capitolo 3 e 4 mostreranno invece l'architettura dell'applicazione, le classi principali progettate e le linee guida seguite per realizzare l'applicazione Java. Nel capitolo 5 si illustreranno infine alcuni esempi pratici di funzionamento dell'applicazione.

SCIFF Framework

Introduzione a SCIFF

SCIFF è un framework logic-based creato all'interno del progetto SOCS con lo scopo di specificare protocolli di interazione tra diverse entità in modo dichiarativo o, più in generale, fornire supporto per lo svolgimento di diversi compiti di verifica.

Il modello di interazione che le entità devono tenere è quindi specificato in modo dichiarativo esprimendo solo vincoli sul comportamento osservabile dall'esterno, senza indicarne le ipotesi sulla loro architettura interna. Tale modello prevede inoltre che le entità interagenti possano comportarsi liberamente se non sono presenti vincoli espliciti.

Questa caratteristica ci ha spinto a scegliere SCIFF all'interno del nostro progetto poiché permette all'utente di muoversi come e dove preferisce, tranne nel caso in cui siano presenti vincoli espliciti.

I concetti fondamentali utilizzati dalla SCIFF per specificare il modello di interazione sono:
- Eventi osservabili che si verificano in fase di esecuzione
- Aspettative su altri eventi e corsi di interazione
- Vincoli di integrità sociale che consentono all'utente di limitare l'interazione globale

Eventi e aspettative

La definizione di evento dipende fortemente dal dominio di applicazione e dal livello di dettaglio desiderato. Il linguaggio SCIFF quindi astrae completamente dal problema di decidere "cosa è un evento", lasciando agli sviluppatori il compito di stabilire quali sono gli eventi importanti per la modellazione del dominio.

Gli eventi sono rappresentati come un atomo

```
H(Event, Time).
```

dove Event è un termine e Time è un numero intero o reale, che rappresenta l'istante in cui l'evento è accaduto.

Nel progetto abbiamo associato a ciascun evento la presenza del giocatore all'interno di determinata cella. Ad esempio l'evento

```
H(sono(1,3,yellow,flower), 4.0).
```

indica che al tempo T=4.0 il giocatore si trova sulla cella di coordinate 1,3. La cella inoltre è di colore yellow e contiene un oggetto flower: tali informazioni sono ricavate in modo univoco dalla rappresentazione del mondo, cioè della scacchiera. HAP è l'insieme di tutti gli eventi che si sono verificati durante l'esecuzione. L'insieme di questi eventi costituisce l'execution trace.

Oltre alla rappresentazione esplicita di cosa è accaduto in un determinato istante, è possibile esprimere in modo esplicito anche una aspettativa sugli eventi che accadranno e sul tempo in cui ci si aspetta che accadano. Il concetto di aspettativa svolge un ruolo chiave per la definizione di protocolli di interazione e più in generale di qualsiasi processo dinamico in continua evoluzione. Risulta, infatti, molto semplice pensare a tali processi in termini di regole nella forma "se A è successo, allora è previsto che accada B". SCIFF supporta anche il concetto di aspettativa negativa, cioè permette di esprimere cosa ci si aspetta non accada.

Per esprimere un'aspettativa positiva si utilizza la seguente sintassi:

```
E(Event, Time).
```

dove Event e Time possono essere variabili o essere valori ground specifici.
Si possono inoltre inserire ulteriori vincoli sulle variabili presenti nell'aspettativa, ad esempio tramite Time > 10.0 si indica che ci si aspetta un Event ad un tempo superiore a 10.0, cioè che l'evento accada dopo l'istante di tempo 10.0.

Al contrario, le aspettative negative sugli eventi sono espresse con

EN(Evento, Time).

In generale, la quantificazione delle variabili all'interno di eventi e delle aspettative positive e negative mantiene il significato più intuitivo: un evento accaduto rappresenta una "classe" di possibili eventi che si verificano, quindi le variabili utilizzate in un evento di successo sono quantificate universalmente. Ad esempio nel nostro progetto, l'evento

H(sono(_,_,yellow,_), T) ∧ T < 5.0.

indica tutte le mosse fatte su caselle di coordinate qualsiasi ma di colore yellow ad un tempo minore di cinque unità di tempo. Le aspettative positive sono invece quantificate esistenzialmente: l'aspettativa è detta *fufilled* quando si verifica il primo evento che la soddisfa.
Quindi tramite

E(sono(_,_,yellow,_), T) ∧ T < 5.0.

significa che dovrebbe esistere una mossa su una cella di colore yellow, di coordinate qualunque, ad un tempo T inferiore a 5. Infine le aspettative negative sono quantificate universalmente, ad esempio

EN(sono(_,_,yellow,_), T) ∧ T < 5.0.

indica che nessuno può muoversi su una cella di colore yellow, di coordinate qualunque, ad un tempo T inferiore a 5.

Vincoli di integrità sociale

I vincoli di integrità sociale (ICS) sono regole usate per mettere in relazione gli eventi già successi e le aspettative future.
Tramite ICS l'utente può quindi limitare l'interazione globale, a partire da una situazione precedente che può essere rappresentata in termini di eventi già accaduti.

I vincoli sono rappresentati come le regole forward nella forma del Body → Head, dove il Body può contenere congiunzioni di letterali e eventi e Head può contenere congiunzioni di aspettative.

Ad esempio

```
H(start,0.0)
/\ H (sono(_,_,yellow,_),T)
--->E (sono(_,_,_,flower),T2) /\ T2 > T.
```

indica che dopo essere stati su una cella di colore yellow in un certo istante T, ci si aspetta di trovarsi su una cella contenente un flower ad un instante T2 tale che T2>T.

Regole SCIFF e regole Prolog

Il programma sviluppato prevede che sia l'utente ad immettere le regole del gioco. In particolare tali regole possono essere inserite in due differenti modalità, le quali non sono mutuamente esclusive ma possono essere impiegate allo stesso tempo.
La prima modalità prevede l'immissione diretta di regole in linguaggio SCIFF. Di seguito sono riportati alcuni esempi di regole con relativa spiegazione.

Il gioco deve terminare entro 7 messe

```
H(start,0.0)
--->EN(sono(_,_,_,_),T2) /\ T2 > 7.0.
```

Se il giocatore passa sui soldi deve andare sul tesoro entro 3 mosse

```
H(start,0.0)
/\ H(sono(_,_,_,money),T)
--->E(sono(_,_,_,treasure),T2) /\ T2 > T /\ T2 <= T+3.0.
```

Se il giocatore passa su flower NON deve passare su paper per le prossime due mosse

```
H(start,0.0)
/\ H(sono(_,_,_,flower),T)
--->EN(sono(_,_,_,paper),T2) /\ T2 > T /\ T2 <= T+2.0.
```

Se il giocatore passa su una cella gialla deve passare su una cella rossa posta a sinistra entro 2 mosse

```
H(start,0.0)
/\ H(sono(X,_,yellow,_),T)
--->E(sono(X2,_,red,_),T2) /\ T2 > T /\ T2 <= T+2 /\ X2 #< X1.
```

Poiché non è richiesta la conoscenza specifica del linguaggio SCIFF da parte dell'utente, è stata data anche la possibilità di scrivere anche regole in Prolog.
Tale sistema permette all'utente di usufruire di una lista di regole predefinite, le quali possono essere personalizzate impostando gli opportuni parametri.
In pratica sono state scritte alcune regole SCIFF in modo tale che richiamassero un predicato Prolog: sfruttando il meccanismo dell'unificazione si raggiunge quindi un certo grado di flessibilità.

Di seguito sono quindi alcuni esempi di regole predefinite insieme ai relativi casi d'uso.

Specifica della cella iniziale e della cella di arrivo

Regola SCIFF:

```
H(start,0.0)
/\ begin(X, Y, C, F)
--->E(sono(X, Y, C, F), 0.0).

H(start,0.0)
/\ end(X, Y, C, F)
--->E(sono(X,Y,C,F),T) /\ T > 0.0
/\EN(sono(_,_,_,_),T2) /\ T2 > T.
```

Esempi di regole Prolog (che saranno opportunamente inserite dall'utente):
Cella iniziale deve essere di coordinate 1,1: begin(1,1,_,_).
Cella iniziale deve essere di colore yellow: begin(_,_,yellow,_).
Cella di arrivo deve contenere una home: end(_,_,_,home).

Vincolo sul minimo/massimo numero di mosse

Regole SCIFF

```
H(start,0.0)
/\ minmove(MINT)
--->E(sono(_,_,_,_),T2)
/\ T2 > MINT.

H(start,0.0)
/\ maxmove(MAXT)
--->EN(sono(_,_,_,_),T2)
/\ T2 > MAXT.
```

Esempi di regole Prolog:
Il gioco deve terminare al massimo in 5 mosse: maxmove(5.0).
Il gioco deve durare almeno 2 mosse: minmove(2.0).

Vincolo sulla masso successiva

Regola SCIFF

```
H(start,0.0)
/\ force([X,Y,C,F],[X2,Y2,C2,F2], DELAY)
/\ H(sono(X,Y,C,F),T)
--->E(sono(X2,Y2,C2,F2),T2)
/\ T2 > T
/\ T2 <= T+DELAY.
```

Esempi di regole Prolog:
Dopo essere stato sulla cella di coordinate 2,3 il giocatore deve andare su una cella di colore red entro 5 mosse:
force([2,3,_,_], [_,_,red,_], 5.0).

Dopo essere stato su una cella di colore COL contenente una money, il giocatore deve andare su una cella dello stesso colore contenente un treasure entro 3 mosse:
force([_,_,COL,money], [_,_,COL,treasure], 3.0).

Vincolo sulla direzione di spostamento

Stabilisce come il giocatore deve spostarsi rispetto all'ultimo click, per un certo numero di mosse

Regola SCIFF

```
H(start,0.0)
/\ move_rules([X,Y,C,F], COND, DELAY)
/\ H(sono(X,Y,C,F),T)
-->E(sono(X2,Y2,C2,F2),T2)
/\ T2 > T
/\ T2 <= T+DELAY
/\ call(COND, X, Y, X2, Y2).
```

Esempi di regole Prolog:
Dopo essere stato su un paper, il giocatore deve spostarsi verso sinistra per le successive 2 mosse: move_rules([_,_,_,paper], left, 2.0).
Dopo essere stato su una cella di colore yellow, il giocatore deve spostarsi verso l'alto per le successive 4 mosse: move_rules([_,_,yellow,_], up, 4.0).

Nota: i predicati left e up sono anch'essi immessi dall'utente. Ad esempio è possibile indicare il predicato left come segue:

```
left(X1,Y1,X2,Y2) :- X2 #< X1.
```

Tale predicato, come si vede dalla regola SCIFF, viene invocato tramite una call passando le coordinate del giocatore in modo che possano essere eseguiti gli opportuni controlli.
In questo modo l'utente può specificare qualunque regola di movimento: movimento verso l'alto, verso sinistra, in diagonale, ecc...

Come utilizzare SCIFF

In questo paragrafo saranno fornite alcune indicazioni pratiche per poter utilizzare correttamente il framework SCIFF. Un progetto SCIFF comprende diversi file, contenuti all'interno di una stessa cartella.

Il file project.pl contiene, tra le altre cose, i riferimenti agli altri file.
Ad esempio:

```
history_file('trace.txt').
ics_file('rules.txt').
sokb_file('kb.pl').
```

indica che per nel progetto corrente è presente:
- un file trace.txt contenente la storia degli eventi osservati
- un file rules.txt contenente i vincoli di integrità sociale
- un file kb.pl contenente una base di conoscenza Prolog

Prima di utilizzare la SCIFF è necessario inserire all'interno del file defaults.pl il percorso della cartella contenente tutti i progetti.
Ad esempio:

```
default_dir('/Users/utente/SciffProjects/').
```

Successivamente, per poter iniziare ad utilizzare la SCIFF, ci si deve posizionare nella cartella che la contiene ed invocare il predicato

```
compile(sciff).
```

Infine si seleziona il progetto su cui lavorare, invocando il predicato

```
project(nomeProgetto).
```

A questo punto è possibile verificare se una traccia è corretta, cioè se rispetta i vincoli specificati, invocando semplicemente

```
run.
```

Per poter sfruttare la SCIFF per generare una traccia è necessario impostare l'opzione fulfiller a on con il seguente predicato

```
set_option(fulfiller, on).
```

In questo modo si attiva la versione generativa della SCIFF, chiamata appunto g-SCIFF. Per recuperare una lista di eventi si può quindi procedere con l'invocazione del predicato

```
run, findall_constraints(h(_,_,_), L).
```

che ritorna in L una lista corretta di eventi.
Tutti i comandi descritti in questo paragrafo vengono invocati all'interno del programma Java sfruttando la libreria JPL descritta nel capitolo successivo.

Introduzione a JPL

JPL è un insieme di classi Java e di funzioni C che forniscono un'interfaccia tra Java e Prolog. JPL utilizza la Java Native Interface (JNI) per connettersi a un motore Prolog attraverso la Prolog Foreign Language Interface (FLI), che è in procinto di essere standardizzata in varie implementazioni di Prolog. JPL non è quindi una pura implementazione di Prolog in Java, ma ricorre ad altre implementazioni di Prolog preesistenti sulle piattaforme supportate. Ad esempio la versione corrente di JPL funziona solo con l'implementazione SWI-Prolog.

Attualmente, JPL supporta solo l'inserimento di un motore Prolog all'interno della Java Virtual Machine. Le versioni future potranno anche sostenere l'incorporamento di una Java VM in Prolog, in modo da poter sfruttare la potente strutturazione in classi fornita dall'ambiente Java all'interno di Prolog.

JPL è progettato in due strati, una interfaccia a basso livello per la FLI Prolog riservata ai programmatori C e un'interfaccia di alto livello Java. Questa secondo interfaccia è destinata in particolare ai programmatori che non vogliono occuparsi nei minimi dettagli di FLI Prolog, ed è quella che abbiamo impiegato nel nostro progetto.

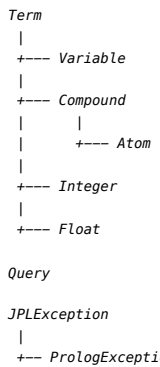
JPL è rilasciato sotto i termini della Gnu Library Public License.

High-Level Interface

JPL 3.0.1 Java-calls-Prolog API fornisce un insieme di classi che nascondono quasi tutti i dettagli nell'interfaccia di basso livello. E' meno flessibile rispetto all'interfaccia a basso livello, ma ha anche una curva di apprendimento più piccola e per molti versi è più naturale e Prolog-like rispetto all'interfaccia di basso livello.

Il package Java di JPL contiene tutte le classi di questa interfaccia. Nessuna delle classi corrisponde a uno dei tipi di dati della Prolog Foreign Language Interface (FLI).

API consiste nella seguente gerarchia di classi:



- **Term** è una classe astratta, di conseguenza solo le sue sottoclassi possono essere istanziate.
- Ogni istanza di **Query** contiene almeno un termine che indica l'obiettivo che deve essere dimostrato (goal).
- Ogni istanza di **Compound** è un nome (java.lang.String) e un array di argomenti (Term) obbligatoriamente non vuoto.
- **Atom** è una specializzazione di Compound con zero argomenti.
- **Variabile** sono individuate da nomi, i quali devono rispettare la sintassi del testo convenzionale sorgente Prolog. Quindi:

```
Variable X = new Variable("X"); // a regular variable
Variable X = new Variable("_"); // an "anonymous" variable
Variable X = new Variable("_Y"); //a'dont-tell-me'var,whose bindings we don't want to know
```

Querying Prolog

Per eseguire una Query Prolog tramite l'interfaccia di alto livello, prima si costruisce un oggetto Query e poi si utilizza l'interfaccia java.util.Enumeration, implementata dalla classe Query stessa per ottenere soluzioni. Per "soluzione" si intende quello che è conosciuto in gergo logico di programmazione come una sostituzione, ovvero un insieme di associazioni, ognuna delle quali riguarda una delle variabili all'interno obiettivo della Query.

```
public interface Enumeration {
     public boolean hasMoreElements();
     public Object nextElement();
}
```

Il metodo `hasMoreElements()` può essere utilizzato per determinare se una query ha qualche (o qualsiasi ulteriore) soluzione. L'invocazione del metodo `query.hasMoreElements ()` restituisce true se la query Prolog è dimostrabile, e false altrimenti.
Se il goal della Query contiene variabili, la sua esecuzione porta a una sequenza di binding di queste variabili ai Term. L'interfaccia ad alto livello usa un java.util.Hashtable per rappresentare queste associazioni e gli oggetti nella tabella sono Termini.
Per comodità, la classe Query fornisce altri due metodi con le seguenti firme:

```
public boolean hasMoreSolutions();
public Hashtable nextSolution();
```

Una comodità derivante dall'utilizzo del metodo `nextSolution()` è nel non dover più effettuare un cast esplicito a Hashtable.

Il framework fornisce, inoltre, la possibilità di ottenere una soluzione in quanto spesso ci interessa ottenere solo la prima soluzione trovata. La classe Query, a tal scopo mette a disposizione il metodo

public Hashtable oneSolution();

In caso non esistano soluzioni il risultato restituito è null; invece se la query richiesta è grounded, ovvero non contiene variabili, verrà restituita una Hashtable vuota.
Qualora fosse necessario ottenere tutte le possibili soluzioni, la classe Query offre il metodo:

```
public Hashtable[] allSolutions();
```

Il risultato conterrà tutte le soluzioni di query, nell'ordine in cui sono stati ottenute e come per il comando prolog findall/ 3, i duplicati non sono rimossi. Se la query non ha soluzioni, questo metodo restituisce un array vuoto.
Infine, un altro metodo utilizzato all'interno del progetto e molto utile è semplicemente sapere se la query è dimostrabile, non quali soluzioni ha; ed ecco il metodo

```
public boolean hasSolution();
```

Questo metodo è equivalente, ma spesso più efficiente, della chiamata a oneSolution() e verificare se il valore di ritorno non è nullo, cioè se la query ha avuto successo.

Eccezioni

Il pacchetto JPL prevede la gestione delle eccezioni abbastanza grezza. La classe di base per tutte le eccezioni JPL è `JPLException`, che è un `java.lang.RuntimeException` (e quindi non deve essere dichiarata), e che sarà lanciata in assenza di qualsiasi altro tipo di eccezione che possono essere lanciate, solitamente come risultato di qualche errore di programmazione. Convertire l'eccezione a un `java.lang.String` dovrebbe fornire alcune informazioni descrittive sul motivo per l'errore. Tutte le altre eccezioni estendono JPLException. Attualmente ve ne sono due: la classe `QueryInProgressException` e la classe `PrologException`.

Una `QueryInProgressException` viene generata quando una query viene aperta mentre un'altra è in corso; questa eccezione può capitare in situazioni multi-thread.

Una *PrologException* può essere lanciata sia durante l'esecuzione di un predicato Prolog built-in, sia da un throw esplicito nel codice Java, sia direttamente dal codice Prolog dell'applicazione attraverso il predicato *throw/1*.

Non vi è attualmente alcuna necessità di gestire delle eccezioni causate da parametri non corretti, ad esempio, predicati non definiti passati attraverso l'interfaccia di alto livello al motore Prolog.

Architettura del sistema

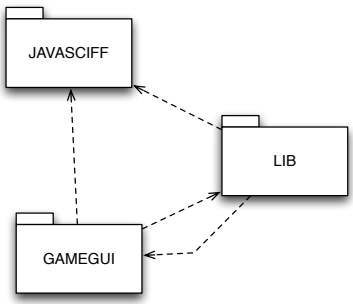


Figura 1. Package

Il sistema è strutturato in tre package:

- **GAMEGUI**: contiene l'implementazione della grafica creata utilizzando il NetBeans Designer e gli handler relativi agli eventi intercettati dall'interfaccia
- **JAVASCIFF** : contiene classi per l'utilizzo via Java della Sciff.
- **LIB**: contiene classi di utilità.

Nei paragrafi successivi saranno mostrati i diagrammi UML di ciascun package, seguiti da una breve descrizione delle classi principali.

Package JAVASCIFF

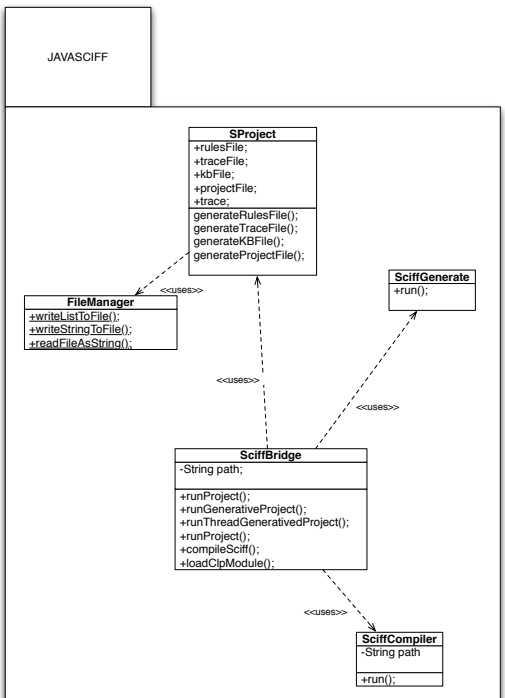


Figura 2. Package JavaSciff

- *SProject* : si preoccupa di creare un progetto conforme: quindi con un file di regole (rulesFile), con un file contenente la traccia di risoluzione (traceFile), la base di conoscenza (kbFile) e il file di progetto (projectFile).

- *SciffBridge*: è il ponte tra Java e SCIFF quindi espone metodi per eseguire progetti, per compilare SCIFF, per caricare il modulo CLP. Questi metodi si incaricano di eseguire i comandi SCIFF relativi e di restituirne il risultato.

- *SciffCompiler*: la compilazione SCIFF è affidata a un'istanza di questa classe che è un thread. Questa esigenza è nata in modo da poter caricare velocemente l'applicazione senza forzare l'utente ad aspettare la compilazione di SCIFF che risulta onerosa.

- *SciffGenerate*: anche il processo generativo della traccia partendo da un numero massimo di mosse definito è affidato a un thread dedicato. Questo per un semplice motivo: è un'operazione lunga e costosa, pertanto si è deciso di dare la possibilità all'utente di effettuare un abort. Per non rendere il sistema sordo alla richiesta dell'utente di interruzione dell'operazione, essa viene eseguita da un thread esterno sul quale il processo principale può lanciare un signale di kill, forzandone la terminazione.

- *FileManager*: espone semplicemente funzioni che occorrono alla generazione del progetto. Offre la possibilità di scrivere liste (tracce) o stringhe su file e la loro lettura. La lettura è necessaria in quanto si è deciso di utilizzare un progetto di default da presentare all'utente.

Package LIB

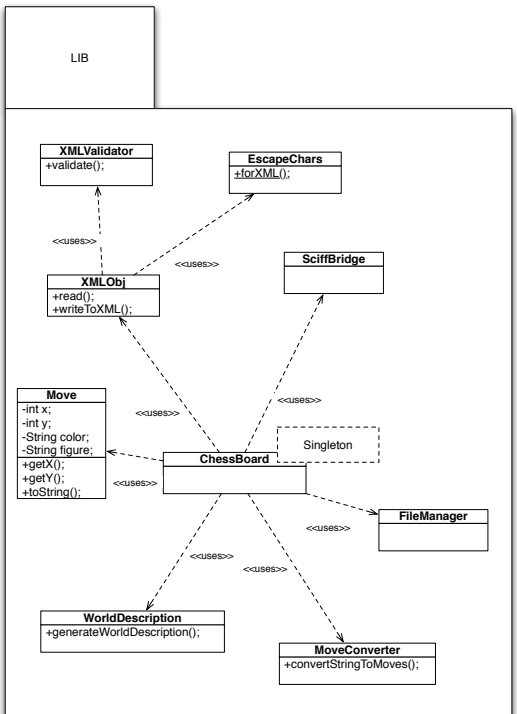


Figura 3. Package Lib

- `ChessBoard`: scritta utilizzando il pattern Singleton in modo da poter recuparne l'istanza in vari punti. Rappresenta la scacchiera di gioco e tutta la relativa interfaccia.

- `XMLObj`: per maggiore flessibilità si è deciso di riempire le celle della scacchiera e le regole leggendone la struttura da un file XML (metodo read()). Questa classe offre anche un metodo di writeToXML() per poter fare un export del mondo in modo da poterlo ricaricare in una successiva partita.

- `XMLValidator`: il sistema verifica che l'XML che si sta provando a caricare sia conforme a un file XSD (schema) che impone un determinato formato.

- `EscapeChars`: quando si esporta la descrizione del mondo e delle regole è necessario sostituire alcuni caratteri con le relative `entity.forXML()` restituisce la stringa in ingresso con le relative sostituzioni.

- `Move`: rappresenta una mossa.

- `MoveConverter`: ha il compito di parsare la traccia generata da SCIFF e di convertirla in una lista di mosse che possa poi essere rappresentata sulla scacchiera.

- `WorldDescription`: genera la descrizione del mondo, dove con mondo si intende una definizione completa della scacchiera e delle relative celle.

Package GAMEGUI

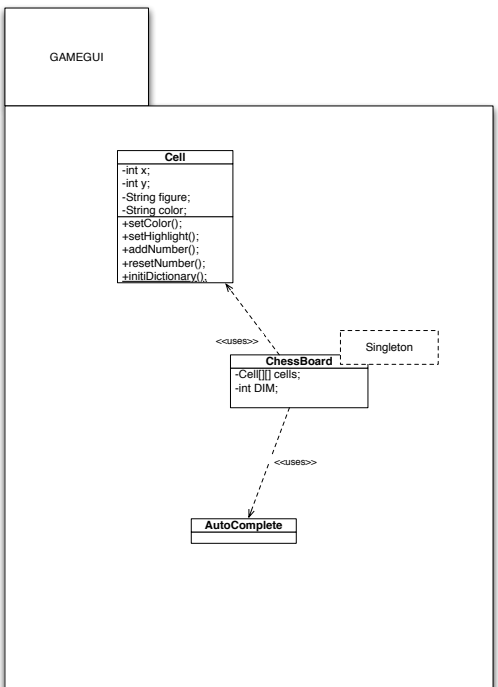


Figura 4. Package GUI

- `Cell`: elemento base della scacchiera. Rappresenta una cella. E' stata creata una classe esterna per poter gestire meglio la complessità del mondo.

- `AutoComplete`: una classe Sun che implementa il completamento automatico delle parole all'interno di un campo di testo tramite una ricerca binaria all'interno di un dizionario.

L’applicazione

In questo capitolo si presenta l’interfaccia dell’applicazione “Sciff Game” ed il suo funzionamento.

Come si può notare dalla figura sottostante l’interfaccia è suddivisa in quattro regioni principali, adibite a funzionalità differenti.

Figura 5. Interfaccia grafica gioco.

Andando da sinistra a destra notiamo:

- La scacchiera, che rappresenta l’ambiente di gioco
- Un editor di regole Prolog
- Un elenco di mosse (*execution trace*)
- Un editor di regole SCIFF

La scacchiera è completamente configurabile grazie al file xml che ne descrive caratteristiche e contenuto. Inoltre tale file può essere caricato dinamicamente anche quando l’applicazione è già stata avviata, in modo tale da consentire di realizzare differenti ambienti di gioco.
La scacchiera consente una facile interazione con l’utente che può specificare il percorso risolutivo cliccando sulle singole caselle che la compongono, contemporaneamente si vedranno comparire la rappresentazione testuale del percorso selezionato sulla scacchiera nell’elenco di mosse.

Figura 6. Percorso evidenziato.

Le regole sono inserite in un editor testuale e possono essere modificate a piacimento; inoltre possono essere memorizzate su file e successivamente richiamate.

Le regole scritte in Prolog, come già detto, sono specifiche e definite per una singola scacchiera e di conseguenza sono salvate con la stessa nel file xml che la definisce; al contrario delle regole SCIFF sono generiche e indipendenti, anche se per queste è stato pensata la possibilità di un caricamento dinamico.

Sono stati definiti due possibili funzionamenti: in un caso è l'utente che deve cercare la soluzione corretta, nell'altro è l'elaboratore che deve generare un percorso congruente con le specifiche.
Nel primo caso l'applicazione svolge solo il ruolo di verifica del percorso e se questo è corretto l'utente ha completato il gioco.
Nel secondo caso invece l'utente deve solo specificare quante mosse deve essere lungo il percorso risolutivo da generare e l'elaboratore farà il resto; la soluzione proposta comparirà come sequenza di mosse sia sulla scacchiera sia nell'elenco testuale.

Verifica del percorso

Dopo aver caricato la configurazione desiderata della scacchiera e delle regole Prolog l'utente deve trovare una soluzione.
Si consideri ad esempio la configurazione della scacchiera precedentemente mostrata e le seguenti regole prolog:

```
begin(1,1,_,_).
end(_,_,_,home).
maxmove(10.0).
force([_,_,_,money], [_,_,yellow,treasure], 3.0).
force([_,_,_,treasure], [_,_,blue,paper], 3.0).
move_rules([_,_,_,paper], left, 1.0).
left(X1,Y1,X2,Y2) :- X2 #< X1.
prereq([_,_,_,home], [_,_,blue,paper]).
```

L'utente immette il percorso cliccando sulle caselle della scacchiera. Non è previsto nessun meccanismo di verifica real-time della soluzione. Al momento non è quindi possibile verificare se il percorso immesso è corretto finché questo non viene completato.

Non viene, inoltre, effettuata nessuna verifica sull'esistenza effettiva di una soluzione. Alcune regole potrebbero essere in contrasto e potrebbe non esistere nessuna soluzione corretta. Deve essere compito di chi scrive le regole assicurarsi che queste non generino conflitti e che il gioco sia effettivamente risolvibile.

Nel caso mostrato nell'esempio precedente, una possibile soluzione è data dal seguente percorso:

```
hap(sono(1,1,blue,money), 0.0).
hap(sono(5,2,yellow,treasure), 1.0).
hap(sono(1,4,blue,paper), 2.0).
hap(sono(0,5,default,empty), 3.0).
hap(sono(1,5,default,empty), 4.0).
hap(sono(7,7,default,home), 5.0).
```

Dopo averlo inserito nell'applicazione, l'utente può quindi verificare la correttezza del cammino individuato cliccando sul bottone "Verify Trace"**.**

Generazione del percorso

Per quanto riguarda la generazione del percorso abbiamo utilizzato la funzionalità generativa della SCIFF, g-SCIFF. L'approccio utilizzato però non risulta ottimizzato poiché opera come un algoritmo *Generate-and-Test,* producendo un albero di ricerca molto vasto e richiedendo quindi un ampio utilizzo di risorse durante l'esecuzione.
Per cercare di ottimizzare il più possibile l'esecuzione sono state utilizzate delle regole SCIFF specifiche per la scacchiera caricata, senza che fossero mediate da predicati prolog.

```
// 1 Regola per avere le variabili ground nelle celle
H(sono(X,Y,C,F),_)
--->cell(X,Y,C,F).
```

La regola, sopra riportata, è indispensabile al funzionamento della g-SCIFF e quindi per la generazione di un percorso corretto; essa ha infatti il compito di trasformare le variabili `(X,Y,C,F)` nei relativi valori ground. Essa impone che il percorso risolutivo appartenga alla scacchiera del gioco; per fare ciò è stato necessario ampliare la nostra base di conoscenza e inserirvi, come "*fatti*" l'intera descrizione della scacchiera. Di seguito si può vedere una sezione della base di conoscenza, che è realizzata in modalità dinamica e creata ex-novo prima della generazione di un nuovo percorso.

...
```
cell(0,0,yellow,arm).
cell(0,1,default,empty).
cell(0,2,blue,money).
cell(0,3,default,empty).
cell(0,4,default,empty).
cell(0,5,default,empty).
cell(0,6,yellow,star).
cell(0,7,yellow,book).
cell(1,0,red,paper).
cell(1,1,blue,money).
```
...

Questo tipo di regola è fortemente responsabile delle scarse performance legate alla fase di generazione del percorso, essa infatti ad ogni tempo T (ogni passo di soluzione) fa matching con l'intera base di conoscenza facendo crescere a dismisura lo spazio di soluzione, che viene raffinato dalle regole successive.

```
// 2 Cella iniziale (0,0,yellow,arm)
H(start,0.0)
--->E(sono(0,0,yellow,arm),0.0).

// 3 Cella finale (4,4,default,empty)
H(start,0.0)
--->E(sono(4,4,default,empty),T) /\ T > 0
/\ EN(sono(_,_,_,_),T2) /\ T2 > T.
```

Queste due aspettative (2, 3) indicano quale sia la casella di partenza e quella d'arrivo assegnandogli coordinate, colore e figura.

```
// 4 Non puoi restare fermo nella stessa cella
H(start,0.0)
/\ H(sono(X,Y,C,F),T)
/\ H(sono(X,Y,C,F), T2) /\ T2==T+1.0
--->false.
```

```
// 5 Se passo su arm DEVO andare sulla casella rossa con il troll entro 3 mosse
H(start,0.0)
/\ H(sono(_,_,_,arm),T)
--->E(sono(_,_,red,troll),T2) /\ T2 > T
/\ T2 <= T+3.

// 6 Se passo su una casella rossa DEVO andare su una blue entro 3 mosse
H(start,0.0)
/\ H(sono(_,_,red,_),T)
--->E(sono(_,_,blue,_),T2) /\ T2 > T
/\ T2 <= T+3.
```

Le regole sopra indicate (4, 5, 6) impongono ulteriori vincoli sul percorso: la prima impone che non si possa restare fermi sulla stessa cella per tempi consecutivi, mentre le altre due obbligano a transitare su caselle con determinati attributi.

Prima di proseguire con la generazione del percorso è necessario specificare il numero di mosse con il quale si desidera completare il cammino; questa specifica si è resa necessaria per la realizzazione di una traccia risolutiva compatta nei tempi; specificando il numero di mosse viene generata una traccia generica le cui variabili (`A,B,C,D`) assumeranno il valore degli attributi delle caselle di soluzione.

```
// Traccia generica
hap(sono(A,B,C,D), 0.0).
hap(sono(A,B,C,D), 1.0).
hap(sono(A,B,C,D), 2.0).
hap(sono(A,B,C,D), 3.0).
```

Dopo aver specificato la lunghezza del percorso, premendo il bottone “Generate Trace” questa comparirà sia sulla scacchiera sia come elenco di mosse.
Di seguito è riportata la soluzione generata in automatico dalla g-SCIFF.

```
// Traccia soluzione di un percorso di 4 Mosse
hap(sono(0,0,yellow,arm), 0.0).
hap(sono(2,2,red,troll), 1.0).
hap(sono(0,2,blue,money), 2.0).
hap(sono(4,4,default,empty), 3.0).
```

Come è possibile osservare, sia il percorso soluzione sia il numero di regole SCIFF utilizzate sono numericamente bassi. Questo è stato un accorgimento necessario dettato dalla necessità di ridurre il tempo di esecuzione di questa particolare modalità di gioco. Inoltre è stata adottata una scacchiera di dimensioni ridotte (4x4 invece della classica 8x8), sempre per cercare di lenire i problemi di performance di cui è affetto il sistema.
Anche con tutti questi accorgimenti i tempi di esecuzione risultano inaccettabili, nell'ordine delle decine secondi e le tempistiche peggiorano se si richiede un percorso più lungo; con una traccia di 5 mosse infatti arriviamo a tempi di esecuzione di alcuni minuti.

Conclusioni

Durante questa attività di progetto è stato realizzato un gioco basato su regole comportamentali personalizzabili dall'utente. Le regole sono state modellate attraverso il framework SCIFF e in alternativa possono essere scritte usando semplici predicati Prolog.

Il gioco è in grado di verificare se il percorso immesso dall'utente è corretto. Tale funzionalità risulta perfettamente funzionante e sono state ottenute buone performance anche con l'utilizzo di regole complicate. Inoltre il gioco è in grado di generare un percorso in accordo con le regole inserite, purché la lunghezza del percorso resti limitata.

In realtà uno dei nostri obbiettivi era quello di dimostrare che la macchina riuscisse a battere l'uomo in velocità nel dare una soluzione corretta e coerente a tutte le regole specificate. Purtroppo al momento la funzionalità generativa della nostra applicazione non risulta sufficientemente performante e la sfida contro l'uomo non può ancora ritenersi vinta.

Sono stati infatti riscontrate una serie problematiche legate al tempo necessario per la generazione automatica del percorso da parte del nostro particolare approccio di ricerca (*Generate-and-Test),* il quale richiede un grosso uso di risorse poiché esplora tutte le possibili soluzioni. Anche a fronte della riduzione della scacchiera (da 8x8 a 4x4) il tempo di attesa risulta superiore a quello necessario all'uomo per arrivare a una soluzione corretta.

L'applicazione è stata realizzata sfruttando il framework SCIFF, innovativo e in continua evoluzione, il nostro lavoro vuole rappresentare un buon metodo per valutare le qualità e i futuri sviluppi del framework.