UNIVERSITA' DEGLI STUDI
ROMA SAPIENZA

LAUREA SPECIALISTICA IN INGEGNERIA INFORMATICA

CORSO DI METODI FORMALI NELL'INGEGNERIA DEL SOFTWARE 2007/08

PROFESSOR TONI MANCINI

JASMINE
JAVA SYMBOLIC MODEL INTERPRETER

GENERAZIONE AUTOMATICA DI CASI DI TEST IN UN CAMMINO DI UN GRAFO DI FLUSSO

a cura di:
LUCA PORRINI, CONSTANTIN MOLDOVANU, EMANUELE TATTI

Jasmine (JAva Symbolic Model INtErpreter)

Indice

0	Indice	3
1	Introduzione	4
2	Parsing Java	7
2.1	Panoramica libreria JTB	7
2.2	Da JavaToSMV a Jasmine	8
2.3	Struttura codice Jasmine	15
3	Grafo di flusso	27
3.1	Rappresentazione interna	27
3.2	Chiamate a metodi	30
3.3	Conversione in SMV	32
4	Test cammini	37
5	Integrazione con OOPS	47
6	Interfaccia grafica	49
6.1	Panoramica libreria JGraph	49
6.2	Struttura dell'applicazione grafica	50
6.3	Integrazione con NuSMV	54
7	Conclusioni	57
8	Bibliografia e riferimenti	59

Jasmine - 1. Introduzione

Lo scopo ultimo di questa tesina è la realizzazione di un programma che effettui un parsing di una classe Java e ne crei una opportuna rappresentazione con grafo di flusso dei metodi in essa definiti. Successivamente l'utente potrà scegliere un cammino da percorrere all'interno di un metodo, e sarà compito dell'applicazione convertire il grafo di flusso precedentemente generato in codice smv, che sarà successivamente passato al programma *NuSmv*, il quale a sua volta restituirà come output un caso di test che sia in grado di percorrere il cammino scelto dall'utente.

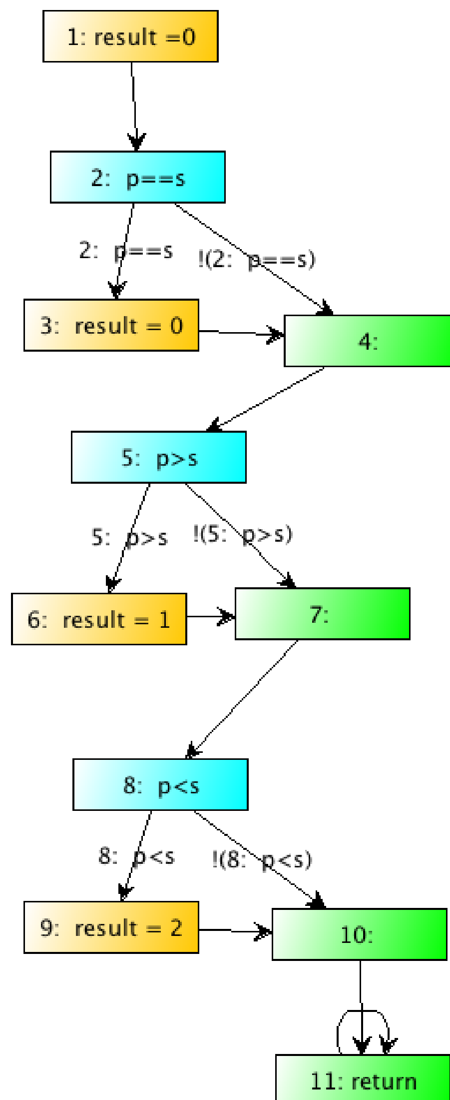
Il nome scelto per la nostra applicazione è **Jasmine**, che sta per *JAva Symbolic Model INtErpreter*. Jasmine realizza, tra l'altro, un'estensione del tool JTB (Java Tree Builder) il cui scopo primario è la verifica della correttezza sintattica di un programma Java. A tal fine, JTB restituisce un albero ad oggetti sintattico, ma NON semantico, per cui non mantiene informazioni sui singoli nodi e archi del grafo di flusso di un metodo Java.

Jasmine riprende in parte e si propone di migliorare il progetto JavaToSMV, iniziato dagli studenti Leone ed Ippoliti, semplificando la creazione del grafo di flusso, aggiungendo il supporto a tutti i possibili costrutti che possono apparire in un programma Java (while, for, do, switch...), e risolvendo alcune incongruenze nella generazione di codice SMV (gestione indici array ed incremento/decremento variabili su tutte). Inoltre, Jasmine ha come obiettivo la generazione di un caso di test che copra un certo cammino scelto dall'utente, mediante una chiamata al tool NuSMV.

A titolo esemplificativo, per rendere più esplicite le potenzialità e le caratteristiche dell'applicazione, analizziamo il comportamento di Jasmine nel momento in cui riceve in input il seguente programma Java, che, dati in input due numeri interi, stabilisce quale dei due è il maggiore (o se sono uguali).

```
public class zeroUnoDue {
    public static int max(int p, int s) {
        int result = 0;
        if (p == s)
            result = 0;
        if (p > s)
            result = 1;
        if (p < s)
            result = 2;
        return result;
    }
}
```

A partire dal codice sopra citato, l'applicazione costruisce un grafo di flusso per il metodo Max così formato:



Quindi l'utente sceglie un cammino da percorrere nel grafo, ad esempio 1 - 2 - 4 - 5 - 6 - 7 - 8 - 10 - 11 (ovvero il caso in cui il primo numero in input sia maggiore del secondo). Jasmine genera il codice SMV relativo alla richiesta dell'utente (permettendogli inoltre di scegliere il range delle variabili che compaiono nel programma, la lunghezza degli array ed eventuali pre e post condizioni), imponendo nella sezione LTLSPEC il passaggio per il cammino desiderato. E' possibile infine lanciare nuSMV sul file SMV generato direttamente all'interno dell'applicazione.
Si otterrà in output un caso di test che percorrerà il cammino precedentemente scelto.

z.p = 1
z.s = 0

Jasmine - 2. Parsing

2.1. Panoramica libreria JTB

Java Tree Builder (JTB) è una libreria utile per generare alberi sintattici di codice Java, utilizzando Java Compiler Compiler (JavaCC). La versione che abbiamo utilizzato è la 1.3.2, la più recente disponibile al momento, che non richiede l'utilizzo diretto di JavaCC ma fornisce un insieme di classi utili per descrivere la struttura sintattica di una qualsiasi classe Java e anche, per completezza, la grammatica con le produzioni utilizzate internamente. Questa versione di JTB supporta versioni di Java fino a 1.4 e non gestisce, quindi, feature più recenti di Java come le annotazioni od i generics.

JTB fornisce, tra altre cose:

- un package "syntaxtree", il quale contiene una classe Java per ogni produzione della grammatica (per esempio ForStatement, Block, ...);
- un parser Java che prende in input una classe Java e restituisce una struttura annidata formata da classi syntaxtree che rispecchiano la struttura del codice Java;
- un package "visitor" che propone un'implementazione del pattern Visitor, DepthFirstVisitor, che consente la navigazione in questa struttura annidata.

Il pattern Visitor è un design pattern comportamentale che permette di separare un algoritmo dalla struttura composta di oggetti cui è aplicato. Il pattern implementa essenzialmente due comportamenti, quello del "visitor", che dichiara un metodo "visit()" per ogni elemento da visitare, e quello dell'"elemento" visitato, che offre un metodo "accept()" nel quale definisce le modalità secondo le quali la visita debba avvenire.
In JTB, ogni classe Java di "syntaxtree", che rappresenta una produzione della grammatica, offre un metodo "accept(visitor)" che non fa altro che invocare il visitor su ogni sottoproduzione in essa contenuta. Per esempio, considerando il costrutto "while", la classe JTB WhileStatement ha la seguente struttura:

```
f0 -> "while"
f1 -> "("
f2 -> Expression()
f3 -> ")"
f4 -> Statement()
```

La classe WhileStatement è costituita da cinque nodi (f0, f1, f2, f3, f4) che possono essere stringhe (f0, f1, f3) oppure ulteriori espressioni Java (f2, f4). Il depth visitor andrà quindi in profondità finquando troverà espressioni da visitare. Utilizzando quest'osservazione abbiamo potuto introdurre le NOP, che verranno descritte in dettaglio in seguito, come operazioni fittizie "eseguite" non appena il visitor termina la visita dell'espressione corrente, sfruttando la struttura intrinsecamente annidata delle classi syntaxtree.

2.2. Da JavaToSmv a Jasmine

Come già citato nel corso dell'introduzione, la prima parte del progetto Jasmine riprende ed estende quanto fatto in passato dalla tesina JavaToSmv, realizzata dagli studenti Leone ed Ippoliti. In particolare, JavaToSmv consentiva la generazione di codice smv a partire da un metodo Java, per poi poterne verificare correttezza parziale e totale. Jasmine si distingue comunque da JavaToSmv sin dalle fasi di strutturazione del codice sorgente, in quanto abbiamo deciso di seguire un approccio ingegneristico e modulare, estendendo quelle classi delle librerie JTB reputate necessarie e non modificarle, come nella tesina precedente, consentendo in questo modo una maggiore separazione del lavoro svolto che si pone come layer aggiuntivo che si appoggia su JTB.

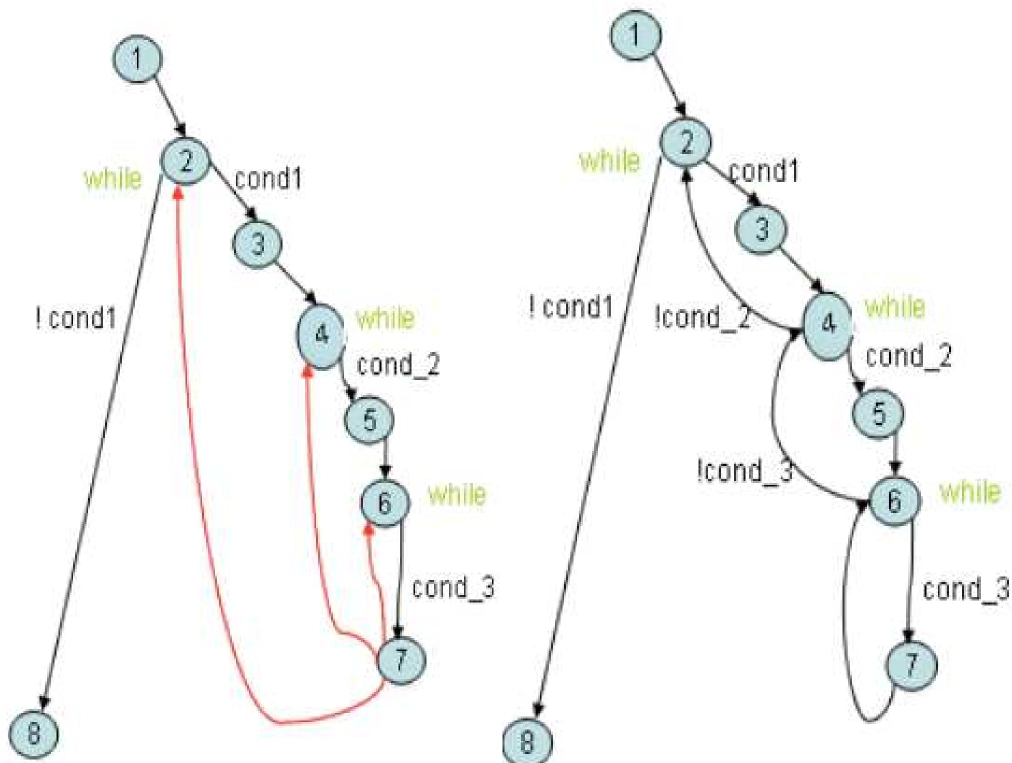
Tuttavia, l'approccio al problema da noi adottato è piuttosto differente rispetto a JavaToSmv: ci siamo proposti di risolvere in maniera più limpida e meno complessa alcune problematiche riscontrate nel precedente programma.

Grafo di flusso:

In primo luogo, una sostanziale differenza è evidente nella creazione del grafo di flusso. Leone e Ippoliti hanno riscontrato un problema nei cicli dei programmi Java, e nella rispettiva traduzione in smv. In particolare, JavaToSmv pativa un problema di ritorno al giusto nodo precedente in particolari casi critici, quali ad esempio dei while annidati.

```
int result = i*j;
    while(result>50){
        result=result-5;
        while(result>100){
            result=result-10;
            while(result>200){
                result=result-20;
            }
        }
    }
return result;
```

Un programma Java come quello sopra citato avrebbe restituito in JavaToSmv il grafo di sinistra (non corretto) anziché quello di destra (corretto):



Per ovviare a questa problematica, Leone e Ippoliti hanno dovuto implementare una serie di "algoritmi offline" che hanno richiesto un nutrito quantitativo di visitors aggiuntivi, che inevitabilmente aumentavano la complessità computazionale e il tempo di generazione del grafo corretto. Ad esempio si è resa necessaria una struttura dati (BackEdgesTable) per la gestione degli archi all'indietro, con una BackEdgeEntry per ogni while visitato. Successivamente viene applicato un algoritmo normalizzatore della BackEdgesTable, che con sofisticati controlli sulla lunghezza degli archi corregge i valori degli init_PC relativi ai salti all'indietro, gestendo in tal modo gli annidamenti complessi. In aggiunta, un ulteriore algoritmo gestisce l'aggiunta di archi all'indietro mancanti nel caso di IF annidati nei while. Un terzo algoritmo gestisce la IfFrontEdgeTable, in cui vengono memorizzate le informazioni sulle transizioni in uscita con archi in avanti nel caso degli ifStatement: tale algoritmo procede alla interrogazione interattiva della BackEdgesTable per verificare l'esistenza di cammini di archi all'indietro con origine in Pc_if+if_length-1, che possa raggiungere un valore PC*<PC_if (caso if con archi all'indietro): nel caso in cui tali cammini non esistano, procede

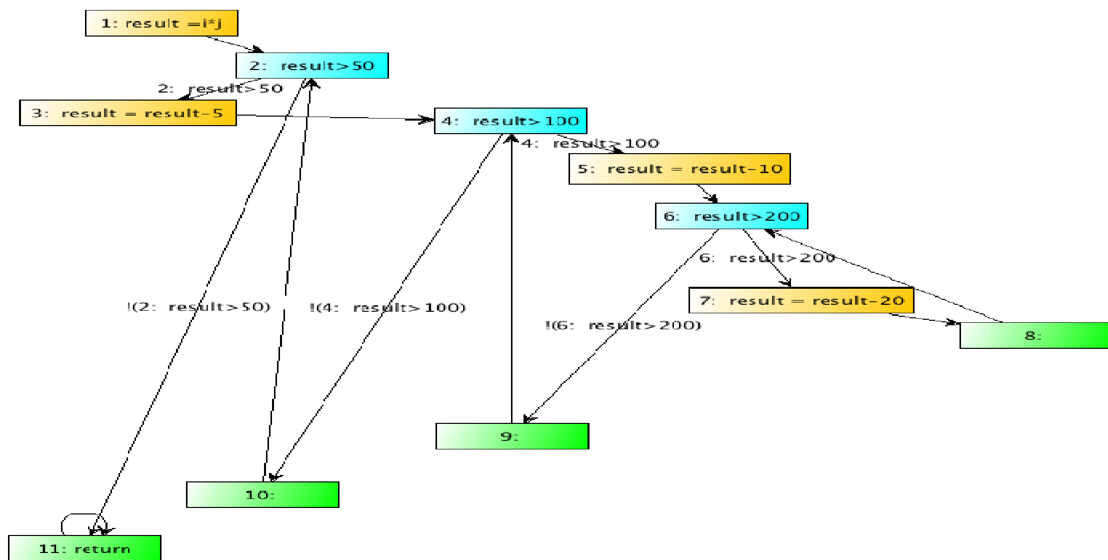
alla creazione di una nuova ifFrontEdgeEntry (arco in avanti). Infine, un LastVisitor procede a sistemare gli esatti valori del Program Counter iniziale e finale, procedendo nuovamente a visitare la BackEdgestTable e la IfFrontEdgeTable.

NOP:

La complessa risoluzione sopra citata non è presente in Jasmine, in quanto l'intero problema non si pone. Per farlo decadere, si è resa necessaria l'aggiunta di un nuovo nodo nel grafo, denominato "*nodo di NOP*" (no operation): a tale nodo si fa corrispondere la fine di ogni blocco (blocco if, while, for...), una scelta che consente all'applicazione di gestire autonomamente e correttamente gli archi all'indietro. Ciò fa sì che il grafo di flusso di un programma Java come quello sopra citato (while annidati) venga correttamente realizzato senza la necessità di alcun algoritmo offline.

Per quanto riguarda la complessità aggiuntiva introdotta dai nodi NOP è rilevante distinguerli in due tipologie:

- NOP di ritorno: questi nodi, imposti alla fine di cicli while (e quindi for, do...while), reindirizzano il flusso di controllo verso il nodo iniziale del ciclo, onde poter rieseguire il ciclo; questo nodo si traduce in un’istruzione SMV di ritorno presente anche nella tesina JavaToSMV, per cui non introduce complessità aggiuntiva;
- NOP di collegamento: questi nodi, imposti alla fine di istruzioni if (con relativo else e alla fine di ogni case di uno switch), hanno l’unico scopo di chiamare il nodo immediatamente successivo all’istruzione if, lasciando inalterate le altre variabili eventualmente presenti; possiamo concludere, quindi, che non appesantiscono in maniera rilevante sulla complessità del codice SMV generato: esse aggiungono un caso nella sezione TRANS il cui unico scopo è incrementare il PC corrente.



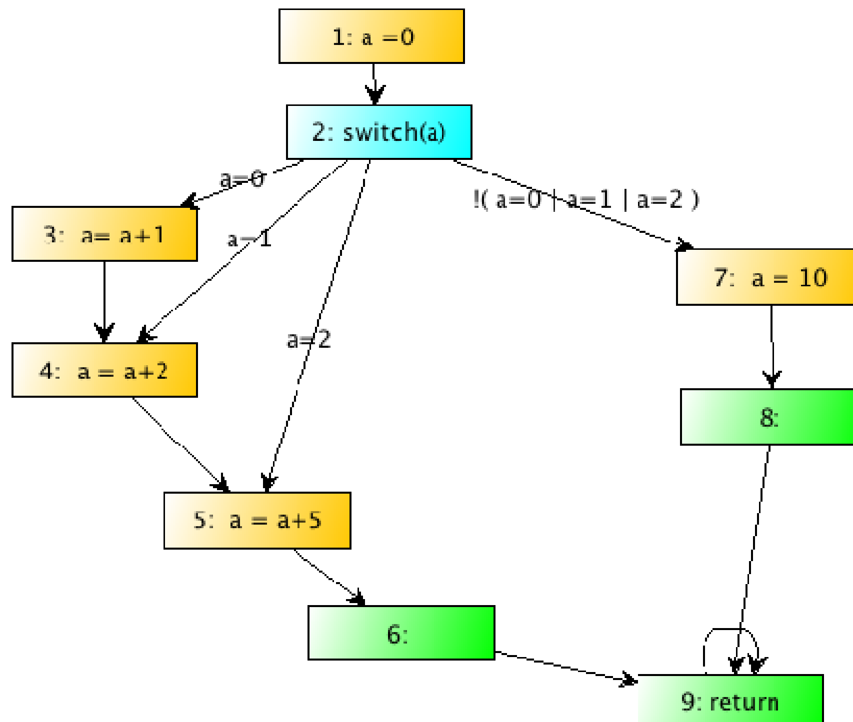
Come si potrà vedere da successivi esempi, tale soluzione può essere estesa ad ogni tipologia di annidamento tra diversi cicli o condizioni, e in ogni caso produrrà in output un grafo semanticamente corretto (anche nel caso in cui un metodo sia privo di parentesi graffe (come ad esempio *if (true) i=5;*).

Cicli e switch:

Rispetto a JavaToSmv, in Jasmine sono stati implementati e sviluppati i casi di ogni tipologia di ciclo che possa presentarsi in un programma Java. Il grafo risultante sarà in grado di rappresentare cicli quali for, do...while e istruzioni complesse come lo switch:

```
public int Switch() {
     int a = 0;
     switch (a) {
          case 0:a++;
          case 1: a=a+2;
          case 2: {
               a = a + 5;
               break;
               }
          default: {
               a = 10;
               break;
               }
```

```
        }
    return a;
}
```



Chiamate a metodi:

Jasmine permette inoltre di riconoscere la chiamata ad altri metodi chiamati da un dato metodo: nel caso in cui il codice del metodo chiamato sia disponibile, un semplice click con il tasto destro sul nodo del metodo chiamante permette di vedere il grafo di flusso del relativo metodo chiamato. Tale funzionalità non è presente in JavaToSmv, che presenta a tal proposito un errore di generazione del file smv nel caso in cui più metodi siano scritti nella stessa classe: in tal caso veniva creato un unico file smv semanticamente scorretto, nel quale si mischiavano dati dei vari metodi (nome del primo metodo con dati di tutti i metodi). Jasmine invece crea automaticamente un file diverso per ogni metodo di una classe, identificandolo con un nome univoco.
Altri problemi riscontrati in JavaToSmv riguardano la traduzione del grafo di flusso nel relativo codice smv. Il codice generato dal programma era pressoché non compilabile, per via di almeno un paio di problemi. In

particolare, nuSmv ha delle limitazioni per quanto riguarda l'indicizzazione degli array, che fanno sì che una condizione del tipo vett[i]>result non possa essere espressa in tal modo: l'indice di un array deve essere una costante, di conseguenza la variabile i non viene accettata. Jasmine ha incorporato un parser che al momento della generazione del codice smv trasforma la condizione, rendendola compilabile. In tal modo, lo stato di TRANS che in JavaToSmv è tradotto con:

```
PC = 4 &  vett[i]>result : next(PC) = 5 & next(result) = result & next(i)
= i ;           -- errore NuSMV a tempo di esecuzione
```

in Jasmine diventa:

```
PC = 4 &  vett[0]>result & i=0 : next(PC) = 5 & next(result) = result &
next(i) = i ;
PC = 4 &  vett[1]>result & i=1 : next(PC) = 5 & next(result) = result &
next(i) = i ;
PC = 4 &  vett[2]>result & i=2 : next(PC) = 5 & next(result) = result &
next(i) = i ;
PC = 4 &  vett[3]>result & i=3 : next(PC) = 5 & next(result) = result &
next(i) = i ;
```

che non dà errori di compilazione (il range di i, che nell'esempio è 3, e la lunghezza dell'array, in Jasmine possono essere scelte dall'utente prima della generazione dell'smv).

La traduzione in smv del grafo di flusso in JavaToSmv presentava inoltre un altro problema, anch'esso fatale per il buon esito dell'esecuzione di NuSmv. Condizioni ed assegnazioni in un dato programma Java erano tradotte esattamente come erano presentate nel codice. Ad esempio, un'istruzione del tipo i++ in Java diventava in smv:

```
PC = 7 : next(PC) = 8 & next(result) = result & next(i) = ++ ;
```

quando invece la corretta rappresentazione in nuSmv è la seguente (adottata da Jasmine):

```
PC = 7 : next(PC) = 8 & next(result) = result & next(i) = i+1 ;
```

Lo stesso problema si presentava per il decremento di una variabile (i--). Un problema simile inoltre riguardava la traduzione dell'uguaglianza in Java: i==1 veniva tradotta, ad esempio, in:

```
PC = 4 & i==1 : next(PC) = 5;
```

che causa anch'esso un errore di esecuzione. In Jasmine il problema è stato corretto, e la condizione sopra citata diventa correttamente:

```
PC = 4 & i=1 : next(PC) = 5;
```

Infine, rispetto a JavaToSmv, Jasmine permette all'utente di indicare (prima della generazione dell'smv) le pre e post condizioni del metodo da valutare: se ne terrà conto al momento della generazione automatica del caso di test.

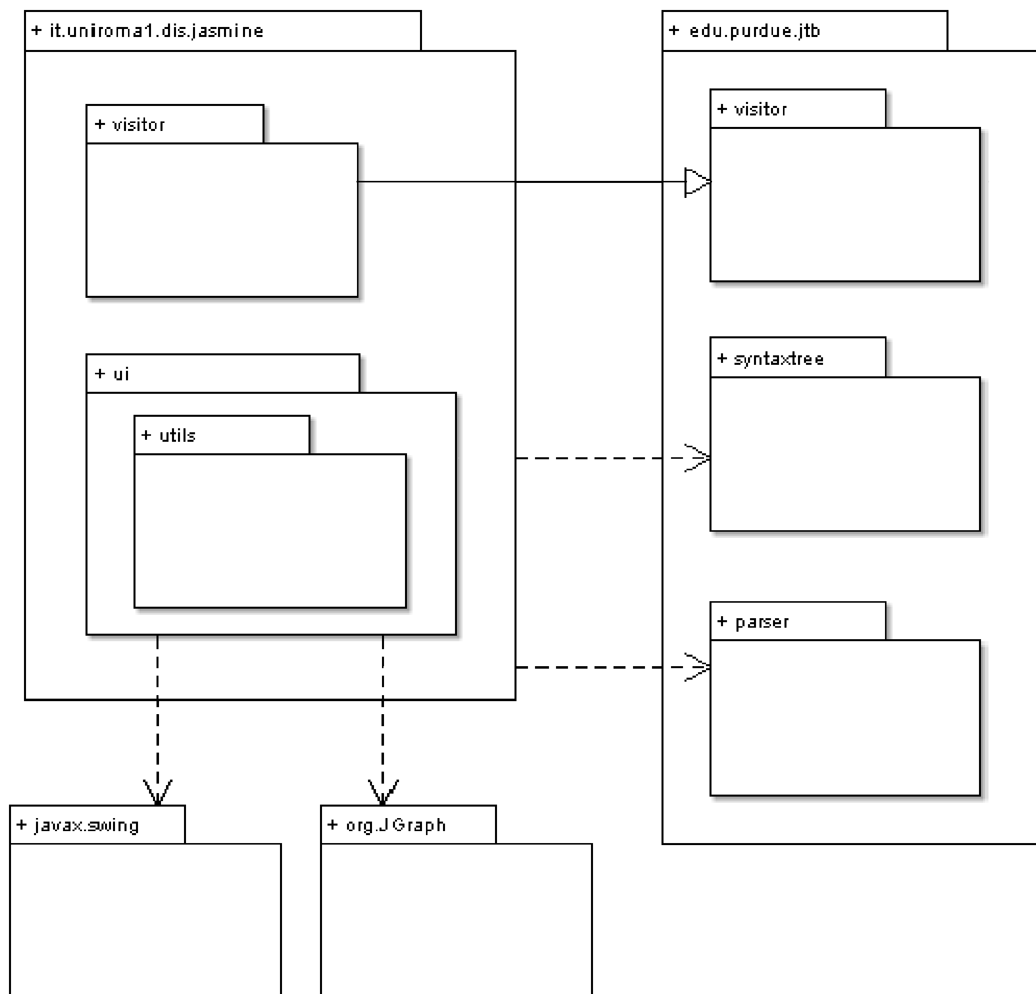
2.3. Struttura codice Jasmine

Struttura dei package:

Jasmine è un'applicazione modulare, organizzata in package che estendono classi presenti in due delle librerie utilizzate: JTB, per parsing codice Java, e Java Swing, per l'interfaccia grafica. Il diagramma successivo mostra le relazioni di *extends* e *uses* presenti tra le classi dei package Jasmine e quelle presenti nelle librerie. Per semplicità di implementazione e comprensione, i files sorgente del package JTB sono distribuiti internamente a Jasmine, ma è possibile includere JTB come libreria esterna, in quanto nessuno dei suoi file è stato modificato.

Nel suo aspetto da componente, Jasmine offre l'interfaccia IJavaParser (*it.uniroma1.dis.jasmine.IJavaParser*), con metodi che consentono un accesso trasparente alle sue funzionalità interne di parsing e generazione codice SMV e casi di test, e l'interfaccia IOperationSpec (*it.uniroma1.dis.jasmine. IOperationSpec*), che realizza l'integrazione con la tesina OOPS.

Nel suo aspetto da applicazione grafica, Jasmine mostra un'interfaccia grafica basata su Java Swing e JGraph che consente un utilizzo delle sue funzionalità. La classe (eseguibile) che ne è punto di accesso è JasmineUi (it.uniroma1.dis.jasmine.ui. JasmineUi).



Struttura dei visitor:

Jasmine utilizza il pattern Visitor, estendendo la classe "EDU.purdue.jtb.visitor.DepthFirstVisitor" di JTB con funzionalità create ad-hoc. La generazione del grafo di flusso della classe passata come parametro avviene attraverso la visita di un solo visitor, TranslatorVisitor; quest'ultimo, internamente, utilizza dei visitor intermedi, che eseguono operazioni di sostegno. I visitor creati sono, quindi:

- TranslatorVisitor: contiene tutta la logica necessaria per estrarre il grafo di flusso dalla classe che sta visitando; sarà discusso in dettaglio in seguito;
- ImageVisitor: chiamato su un qualunque elemento all'interno della classe, ne ricava una descrizione testuale;

- LenghtVisitor: chiamato su un qualunque elemento all'interno della classe, ne calcola la lunghezza in termine di numero do istruzioni;
- MethodVisitor: chiamato all'inizio della visita della classe, ricava i nomi di tutti i metodi presenti al suo interno;

Il TranslatorVisitor effettua override di tutti i metodi reputati utili, realizzando, complessivamente, il seguente percorso all'interno della classe da visitare: all'inizio della classe vengono presi i nomi di tutti i metodi in essa contenuti poi, per ogni metodo, viene creato un grafo di flusso ed effettuata una visita in profondità, inserendo nel grafo corrispondente i nodi corrispondenti alle istruzioni ed eventuali nodi senza operazione (NOP).
In seguito presenteremo quei metodi di TranslatorVisitor ritenuti rilevanti e rappresentativi, raggruppandoli per funzionalità. Nonostante l'apparente semplicità mostrata nei seguenti pseudoalgoritmi, l'implementazione degli stessi ha riscontrato difficoltà, specialmente nel cercare di trovare modalità comuni di ricavare la stessa semantica di codice da forme potenzialmente diverse di sintassi. Per fare un semplice esempio, questi due pezzi di codice sono semanticamente identici:

```
1.  if (a > b) {                  2.  if (a > b)
        c = 2;                            c = 2;
    }
```

mentre strutturalmente sono diversi e per di più nella rappresentazione JTB l'esempio 1 si traduce in una struttura che contiene una classe in più (ossia il blocco di graffe EDU.purdue.jtb.syntaxtree.Block), non presente nell'esempio 2

Espressione if:

La visita di un'espressione if avviene all'interno del metodo
    public void visit(IfStatement n)
Come possiamo osservare dalla definizione della classe IfStatement, essa contiene dei nodi ulteriori:

f0 -> "if"
f1 -> "("
f2 -> Expression()
f3 -> ")"
f4 -> Statement()
f5 -> ["else" Statement()]

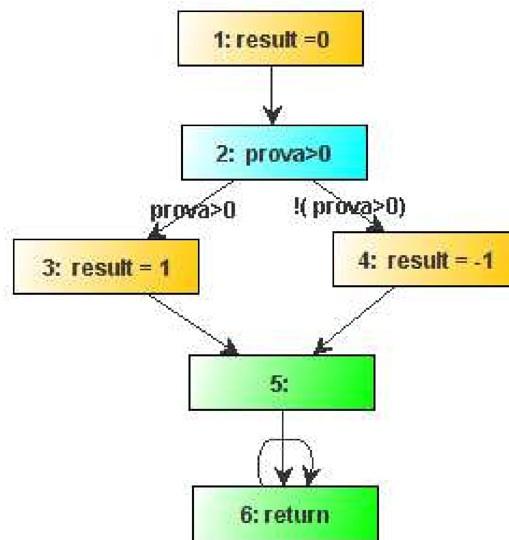
quindi, principalmente, il TranslatorVisitor dovrà effettuare i seguenti passi:

- incrementare il contatore delle istruzioni per considerare anche quest'espressione;
- utilizzare il LengthVisitor per stabilire la lunghezza di quest'espressione ed eventualmente del blocco else, utilizzabile nei successivi reindirizzamenti;
- utilizzare l'ImageVisitor per ricavare l'espressione condizionale presente all'interno del nodo f2;
- generare un nodo che reindirizzi al blocco di istruzioni presenti all'interno dell'if, contenuto nel nodo f4;
- generare un nodo con condizione negata che reindirizzi al blocco di istruzioni successivo all'if (oppure dentro l'eventuale blocco else, contenuto nel nodo f5);
- visitare i nodi f4 ed eventualmente f5, che possono contenere qualsivoglia espressioni annidate;
- finita la visita, generare un nodo fittizio NOP che reindirizzi alla prossima istruzione;

Possiamo notare, quindi, come la stessa struttura annidata delle classi syntaxtree di JTB, assieme ad opportune istruzioni NOP, consente di mantenere la profondità del grafo di flusso.

```
public class TestIf {
    public static int testaIf(int prova) {
        int result = 0;
        if (prova > 0) {
            result = 1;
        }
        else result = -1;
        return result;
    }
```

}



Espressione while:

La visita di un'espressione while avviene all'interno del metodo
public void visit(WhileStatement n)
Effettuata la visita dell'espressione if, abbiamo notato che l'espressione while è concettualmente identica, con l'unica differenza che il nodo NOP finale, invece di reindirizzare all'istruzione successiva, torna alla prima riga del while, effettuando il ciclo.
A tal proposito vediamo la definizione della classe WhileStatement:

```
f0 -> "while"
f1 -> "("
f2 -> Expression()
f3 -> ")"
f4 -> Statement()
```

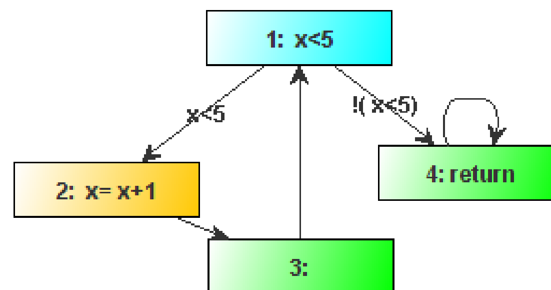
Essa, oltre alla condizione presente in f2, contiene solamente il blocco interno, f4, e non ha la complicazione di un eventuale blocco else. I passi da seguire che si differenziano dal caso if sono, quindi:

- visitare il nodo f4, contenente eventuali istruzioni o espressioni;
- finita la visita, generare un nodo fittizio NOP che reindirizzi alla prima

istruzione del while, mantenuta come variabile;

In questo modo abbiamo realizzato la visita sia dell'istruzione if che di quella while, ottenendo anche un corretto annidamento delle eventuali espressioni interne. Inoltre, dato che ogni altra espressione di Java (for, do...while, switch) può essere idealmente ricondotta ad if o while, possiamo creare i metodi che ne generino i nodi nel grafo di flusso.

```
public class TestWhile {

    public static void testaWhile(int x) {
        while (x<5) {
            x++;
        }
    }
}
```



Espressione for:

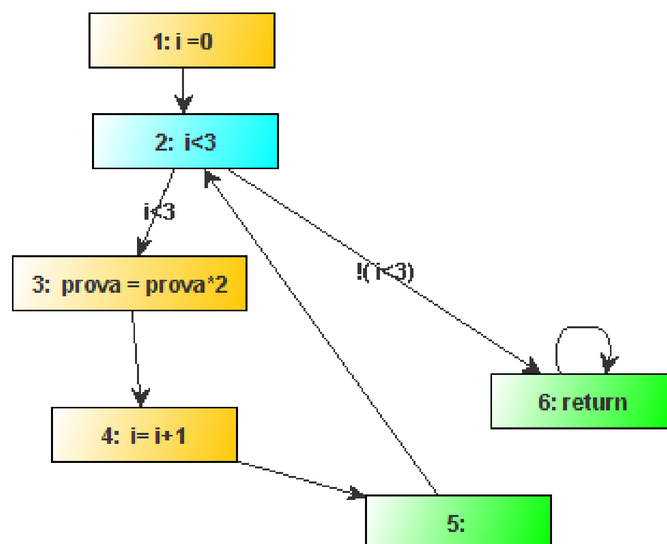
La visita di un'espressione for avviene all'interno del metodo
    public void visit(ForStatement n)
Concettualmente la logica di fondo è molto simile alla casistica del while (così come è simile la semantica dei due cicli in Java).
A tal proposito vediamo la definizione della classe ForStatement:

```
f0 -> "for"
f1 -> "("
f2 -> [ ForInit() ]
f3 -> ";"
f4 -> [ Expression() ]
f5 -> ";"
```

f6 -> [ForUpdate()]
f7 -> ")"
f8 -> Statement()

E' chiaro che un'istruzione del tipo for (int i = 0; i < 5; i++) è equivalente a int i = 0; while (i<5) i++;
Per cui, alla luce di questa uguaglianza, si procede inizialmente con il processare f2, istanziando un nodo con la dichiarazione della variabile.
Quindi si controlla (in un nodo) se l'espressione di f4 è soddisfatta, e nel caso in cui lo sia si procede a valutare lo statement f8 (che può essere qualunque altra istruzione o ciclo Java). Al termine dello statement, un nuovo nodo processa l'update della variabile (f6) ed il ciclo ricomincia, dal momento che un nodo NOP rimanda alla valutazione di f4. E' bene far notare come l'implementazione da noi adottata gestisce anche situazioni atipiche nell'istanziazione di un ciclo for: ad esempio, anche il grafo di for (; ; ;) (senza dichiarazione di variabili e condizioni) è correttamente tradotto.

```
public class TestFor {

    public static void testaFor(int prova) {
        for (int i = 0; i < 3; i++) {
                prova = prova * 2;
            }
        }
}
```

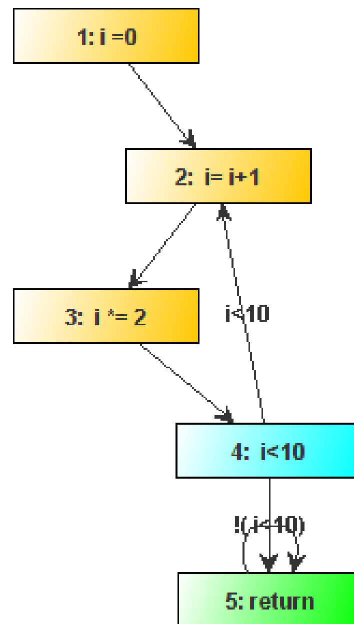


Espressione do...while:

La visita di un'espressione do...while avviene all'interno del metodo
 public void visit(DoStatement n)
Ecco la definizione della classe DoStatement:
 f0 -> "do"
 f1 -> Statement()
 f2 -> "while"
 f3 -> "("
 f4 -> Expression()
 f5 -> ")"
 f6 -> ";"

In questo caso, nel Visitor procediamo prima ad analizzare lo Statement (come da semantica Do), quindi viene aggiunto un nodo che valuta l'espressione del while (f4): nel caso in cui sia soddisfatta, esegue un salto all'indietro fino ad una nuova valutazione dello statement f1, altrimenti procede oltre nell'analisi delle istruzioni successive.

```
public class TestDo {

    public static int doMethod(int[] vett) {
        int i = 0;
        do {
            i++;
            i*=2;
        } while (i < 10);

        return i;
    }

}
```



Espressione switch:

La visita di un'espressione switch avviene all'interno del metodo
public void visit(SwitchStatement n)
Ecco la definizione della classe SwitchStatement:

```
f0 -> "switch"
f1 -> "("
f2 -> Expression()
f3 -> ")"
f4 -> "{"
f5 -> ( SwitchLabel() ( BlockStatement() )* )*
f6 -> "}"
```

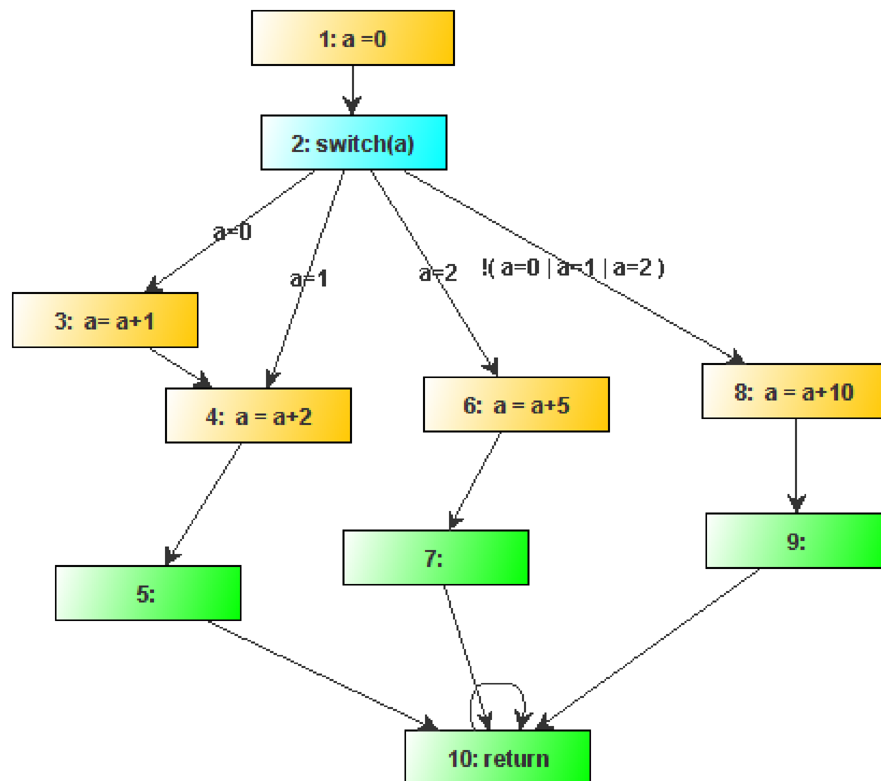
ed ecco il dettaglio di ogni singola SwitchLabel:

```
0 -> "case" Expression() ":"
        | "default" ":"
```

Indubbiamente la valutazione di un'espressione switch è stata tra i problemi più delicati e particolareggiati. Questo anche in conseguenza del fatto che la valutazione di tale ciclo è strettamente legata alla presenza o

meno di istruzioni di break per ogni case: ricordiamo che la presenza di un break causa l'uscita immediata dallo switch, mentre nel caso in cui l'istruzione non sia presente si passa a valutare l'espressione del case successivo. La nostra soluzione prevede la creazione di un nodo iniziale (nodo di switch) dal quale escono tanti archi quanti sono i case (più il default) contenuti nello switch. L'espressione contenuta nello switch è memorizzata in una Stringa variabile, cosi che nella valutazione dei vari casi si possa trasformare un eventuale "case(0)" nell'istruzione corrispondente a "if (i==0)", semanticamente equivalente. L'espressione viene quindi valutata in profondità dal visitor. Nel caso in cui si incontri un'istruzione break, viene creato un nuovo nodo, che reindirizza all'istruzione successiva al termine del blocco dello switch (la lunghezza del blocco di switch è pre-calcolata in profondità dal LengthVisitor e memorizzata in una variabile locale). Altrimenti, nel caso in cui l'istruzione break non sia presente, il nodo è collegato direttamente al successivo, e passa a valutare l'espressione associata alla switchLabel successiva.

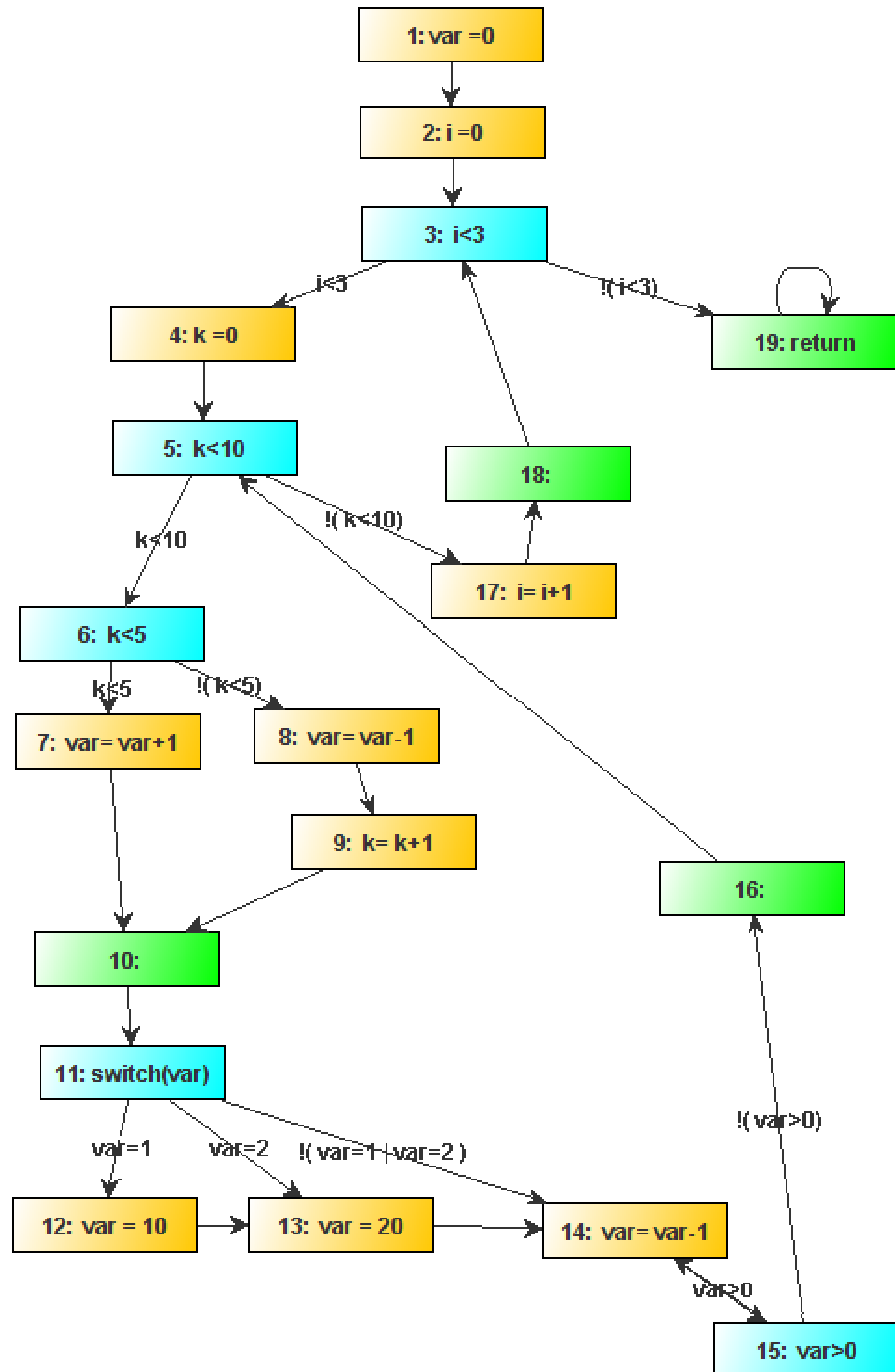
```
public class TestSwitch {

public int Switch() {
int a = 0;
switch (a) {
    case 0:a++;
    case 1:{
        a=a+2;
        break;
    }
    case 2: {
        a = a+5;
        break;
    }
    default: {
        a= a+10;
        break;
    }
}
return a;
}
}
```



Come già accennato in precedenza, i cicli sopra citati possono essere tranquillamente combinati tra di loro, formando in ogni caso un grafo semanticamente corretto. Questo può essere notato anche dal seguente codice java con una quantità esorbitante di annidamenti di diversi cicli, con relativo grafo di flusso mostrato in output da Jasmine.

```
public static void testCaos() {
    int var = 0;
    for (int i = 0; i < 3; i++) {
        int k = 0;
        while (k < 10) {
            if (k<5) var++;
            else {
                var--;
                k++;
            }
            switch (var) {
            case 1: var = 10;
            case 2: var = 20;
            default: do {
                var--;
            }while (var>0);
            }
```

```
            }
        }
    }
```



1: var =0
2: i =0
3: i<3
i<3
!(i<3)
4: k =0
19: return
5: k<10
18:
k<10
!(k<10)
17: i= i+1
6: k<5
k<5
!(k<5)
7: var= var+1
8: var= var-1
9: k= k+1
16:
10:
11: switch(var)
!(var>0)
var=1
var=2
!(var=1 | var=2)
12: var = 10
13: var = 20
14: var= var-1
var>0
15: var>0

Jasmine - 3. Grafo di flusso

3.1. Rappresentazione interna

La rappresentazione interna del grafo di flusso è presa in prestito dal progetto JavaToSmv e aggiornata per poter supportare le nuove funzionalità offerte da Jasmine. Il grafo di flusso è, quindi, costruito tenendo presente la sua futura conversione in codice smv. Essenzialmente, il grafo di flusso è costruito utilizzando le tre classi descritte in seguito, GraphNode, Graph e GraphWrapper.

Classe GraphNode:

La rappresentazione interna del grafo di flusso ruota ovviamente attorno al concetto di nodo del grafo, cui corrisponde un'istruzione del metodo Java considerato. Il nodo del grafo è rappresentato dalla classe "it.uniroma1.dis.jasmine.GraphNode"; essa introduce anche il concetto di PC, il "program counter", il numero dell'istruzione corrente. Andiamo a vedere gli attributi della classe GraphNode:

- initPC - PC dell'istruzione corrispondente al nodo corrente; è un attributo obbligatorio;
- destPC - PC dell'istruzione che seguirà; è un attributo obbligatorio;
- condition - nel caso delle scelte condizionali (if, while, ...), rappresenta la condizione stessa; è un attributo facoltativo;
- variable - nel caso di un'assegnazione di un valore ad una variabile, indica il nome della variabile modificata; è un attributo facoltativo;
- espressione - nel caso di un'assegnazione di un valore ad una variabile, indica il contenuto dell'espressione; è un attributo facoltativo;
- methodCall - presente nel caso in cui l'assegnazione o la condizione viene attuata mediante una chiamata ad un metodo all'interno della stessa classe, contiene il nome del metodo ed altri attributi rilevanti, definiti in seguito; è un attributo facoltativo;
- isSwitch - presente nel caso in cui il nodo corrente rappresenta l'inizio di un'espressione switch; è un attributo facoltativo.

La classe quindi rappresenta un'istruzione nella molteplicità delle sue forme, siano esse assegnazione, condizione, espressioni; inoltre essa effettua l'override del metodo toString(), restituendo una rappresentazione accurata in base alla "forma" del nodo corrente.

Classe Graph:

La classe "it.uniroma1.dis.jasmine.Graph" rappresenta il grafo di flusso di un solo metodo all'interno di una classe; ne viene creato uno per ogni metodo trovato e, man mano che il metodo Java viene visitato, il grafo viene popolato con i dati necessari per rappresentarne la semantica. La classe presenta le seguenti proprietà:

- nodes - vettore di oggetti GraphNode, uno per ogni istruzione; come detto sopra, l'oggetto GraphNode contiene al suo interno informazioni riguardo l'istruzione corrente e la prossima istruzione da seguire, realizzando così la "navigazione" all'interno del metodo Java in causa;
- variables - vettore di tutte le variabili trovate nel metodo, utile poi in fase di generazione codice smv;
- parameters - vettore di tutti i parametri formali trovati nel metodo, utile poi in fase di generazione codice smv;
- numTermIstr - vettore che indica il PC di terminazione delle varie istruzioni, utile poi in fase di generazione codice smv;
- methodName - nome del metodo associato a questo grafo di flusso;
- methodCall – oggetto MethodCall associato a questo grafo;
- loc - numero di istruzione corrente; quando il grafo è generato completamente, coincide con il PC di terminazione del metodo associato.

Classe GraphWrapper:

La classe "it.uniroma1.dis.jasmine.GraphWrapper" è quella che effettivamente viene passata come parametro al TranslatorVisitor e viene utilizzata da quest'ultimo per generare i grafi di flusso di ogni metodo. La classe ha le seguenti proprietà:

- graphMap - un oggetto "java.util.Map" che associa ad ogni metodo

trovato nella classe il suo rispettivo grafo di flusso;
- currentGraph - il grafo correntemente in uso dal TranslatorVisitor;
- currentMethod - il metodo correntemente in uso dal TranslatorVisitor;
- className - il nome della classe Java correntemente visitata.

All'inizio della sua visita su una classe Java, il TranslatorVisitor, oltre alle operazioni già descritte in precedenza, inserisce il nome della classe nell'oggetto GraphWrapper poi visita un metodo alla volta, creando per ognuno di essi un oggetto Graph e riempiendolo di tanti GraphNode quanti reputati necessari per meglio rappresentare il metodo in causa. Tale grafo conterrà nodi sia per le istruzioni ed espressioni trovate sia per le eventuali NOP (no operation) definite al capitolo precedente.

Classe MethodCall:

La classe "it.uniroma1.dis.jasmine.MethodCall" ha un duplice ruolo: se presente come attributo di GraphNode, indica una chiamata a metodo dal nodo rispettivo verso il metodo indicato (attualmente Jasmine gestisce solamente una chiamata a metodo per ogni nodo; vedere la sezione successiva per i dettagli riguardanti la chiamata a metodo); se, invece, è un attributo di Graph, contiene informazioni aggiuntive riguardanti il metodo cui il grafo è associato, come possiamo vedere osservando le sue proprietà:

- methodName – nome del metodo in causa;
- returnType – tipo del valore di ritorno;
- preCondition – stringa contenente la pre condizione associata al metodo;
- postCondition – stringa contenente la post condizione associata al metodo;
- declaredParameters – lista di nomi dei parametri presenti nella dichiarazione del metodo;
- passedParameters – lista di nomi dei parametri passati effettivamente come argomento al momento dell’invocazione; questo attributo è popolato solamente nel caso in cui l’oggetto è attributo di un GraphNode e, quindi, indica quali delle variabili sono

effettivamente passate come parametro.
La classe è creata durante la visita del TranslatorVisitor ed associata sia al grafo corrente che ai vari nodi che effettuano chiamate a metodi, realizzando in pratica il collegamento tra i vari metodi all'interno di una classe. Le sue proprietà vengono poi utilizzate in fase di generazione di codice SMV.

3.2. Chiamate a metodi

Jasmine si distingue dalla precedente tesina, JavaToSmv, anche per il fatto che riconosce le chiamate ad altri metodi della stessa classe e, quindi, ricollega i vari grafi di flusso. Lo scopo di riconoscere le chiamate a metodi è utilizzare le loro pre e post condizioni in fase di test; prendendo un esempio,

```
public static void main() {
    int i = doppio(i);
}

public static int doppio(int n) {
    return n * 2;
}
```

possiamo osservare che il metodo main() contiene un'invocazione a doppio(n). Supponendo che doppio(n) ha come pre-condizione "n è un numero positivo" e come post-condizione "restituisce il doppio di n", possiamo utilizzare la chiamata a tale funzione, in main(), considerandola solamente dal punto di vista delle sue pre e post-condizioni; continuando l'esempio, il codice SMV del metodo main() avrà nella sezione DEFINE:

```
duePRE : n > 0;
duePOST : n * 2;
```

I pre è post cosi ottenuti possono essere utilizzati all'interno del metodo main(), sostituendo la chiamata a metodo; tuttavia possiamo notare che mentre il metodo doppio(n) è stato dichiarato con un parametro di nome n, esso può venir invocato con parametri di nome diverso; Jasmine è in grado di ricavare il nome dei parametri effettivamente passati ad un

metodo e generare correttamente la sezione DEFINE del metodo chiamante, come segue:

```
duePRE : i > 0;
duePOST : i * 2;
```

Notiamo che i è la variabile locale che main() passa come parametro al metodo doppio(n); questo funzionamento si ripete anche nel caso in quale main() passa come parametro una costante e non una sua variabile. E' quindi possibile utilizzare le due variabili SMV appena definite all'interno della sezione TRANS, nei punti in cui si richiede una chiamata a metodo:

```
PC = 1 & duePRE : next(PC) = 2 & next(i) = duePOST;
PC = 2 : next(PC) = 2 & next(i) = i;
```

Questo codice SMV appartiene alla sezione TRANS del metodo main() e possiamo osservare che effettua la chiamata a metodo solamente se la sua pre-condizione è rispettata: in tal caso i assume il valore definito dalla post-condizione del metodo invocato. Si rende necessario contemplare il caso in qui PC = 1 ma la pre condizione duePRE non è soddisfatta; questo può essere gestito aggiungendo un'ulteriore caso in sezione TRANS, dove duePRE è considerata falsa; tuttavia questo approccio potrebbe generare un gran numero di casi in più se le chiamate a metodo sono tante, ragion per cui conviene semplicemente aggiungere un solo caso finale, di default, che contempli il caso in qui una qualunque delle pre condizioni non è soddisfatta. Il codice SMV completo della sezione TRANS del metodo main() è:

```
PC = 1 & duePRE : next(PC) = 2 & next(i) = duePOST;
PC = 2 : next(PC) = 2 & next(i) = i;
1 : next(PC) = PC & next (i) = i;
```

Quest'istruzione di default integra la precedente istruzione di default (che arresta il PC all'ultima riga) consentendo la gestione dei casi nei quali una delle pre condizioni dei metodi chiamati non viene verificata.

In questo modo è possibile utilizzare le pre e post condizioni di metodi invocati come se fossero degli stub "informati" sulle azioni eseguite da questi ultimi metodi. Questi stub sono parametrizzati e consentono

chiamate a metodi con qualunque variabile o costante.

Per quanto riguarda la modalità di inserimento di tali pre e post condizioni, esse, in quanto dipendenti solamente dalla semantica del metodo preso in considerazione, devono essere già specificate in un linguaggio logico, meglio ancora se direttamente in SMV. Per risolvere questo problema, Jasmine offre la possibilità di inserire tali pre e post condizioni da interfaccia grafica, metodo per metodo, assumendo che siano scritte in SMV interpretabile da nuSMV. Il loro contenuto è inteso come:

- pre-condizione: formula logica contenente i parametri in ingresso al metodo; restituisce un valore booleano;
- post-condizione: formula logica contenente i parametri in ingresso al metodo; restituisce un valore dello stesso tipo di quello restituito dal metodo.

Esempio per il metodo int doppio(int n):

```
PRE : n > 0;
POST : n * 2;
```

3.3. Conversione in SMV

La classe Java che si occupa della conversione in SMV del grafo di flusso è FileSmvWriter. In essa sono definiti i metodi generateNuSMV , che chiama un sotto metodo per ogni parte dell'smv (VAR, DEFINE, ASSIGN, TRANS, oltre che a LTLSPEC), ed il metodo writeFileNuSMV, che scrive su un file l'smv creato. Durante la visita dall'albero sintattico di JTB, le informazioni sulle variabili locali e sui parametri formali del metodo vengono memorizzate nelle opportune strutture. In particolare si terrà traccia del nome di ognuna di esse, e del tipo (ad esempio, int o int[]). La stessa visita determinerà quante siano le istruzioni di terminazione del programma (return) con relativi valori del Program Counter, e quale sia il massimo valore del PC raggiungibile, così da limitare la quantità delle righe nella sezione VAR. Il range delle variabili e dei parametri formali è passato al metodo-convertitore tramite un hashTable che fa corrispondere ad ogni variabile/parametro formale il relativo range scelto dall'utente. Nel caso in cui l'utente non abbia specificato alcun range, si utilizzeranno dei valori di default. Quindi un possibile esempio di struttura VAR

generata è il seguente:

```
VAR
 result : 0..3;
 i : 0..3;
 vett : array 0..2 of -10..10;
 PC : 1..9;
```

Nella sezione DEFINE saranno definiti valori di interesse per la successiva valutazione in LTLSPEC: in particolare, è qui che sono definite le condizioni di Terminazione e le Pre e Post condizioni (qualora siano state specificate dall'utente, e passate dall'interfaccia grafica; in caso contrario ci si limiterà a PRE:=1 e POST:=1)

```
DEFINE
 TERM := PC = 9;
 PRE := 1;
 POST := 1;
```

La sezione TRANS rappresenta il cuore della traduzione da grafo di flusso a smv. E' qui che viene mantenuta la logica e la semantica del programma Java di input. L'oggetto Node è assegnato ad ogni riga di TRANS, e popolato durante la creazione del grafo di flusso. In esso saranno memorizzate le informazioni sul PC di partenza, il PC di destinazione, eventuali condizioni di passaggio, ed eventuale modifica ai valori delle variabili o dei parametri formali dovuta a particolari istruzioni della corrispondente riga nel programma Java di input. Si è ricorso inoltre ad un correttore per l'istanziazione degli array (metodo repairVectors): tale correzione modifica istruzioni come vett.length o vett[i], traducendole in codice compilabile tramite NuSMV. In particolare, ogni vett.length viene sostituito con il corrispondente valore relativo alla lunghezza dell'array, mentre per ogni vett[i], si deve iterare l'istruzione e fare in modo che tenga conto di tutti i possibili valori di i, per far si che sia gestito senza problemi il caso relativo al reale valore di i assunto in quell'istante: la riga relativa viene quindi replicata un numero di volte pari al numero di possibili valori che i può assumere.
Per quanto riguarda il side-effect sui vettori, esso viene gestito esattamente come su ogni altra variabile e quindi, case per case, ad ogni elemento del vettore viene assegnato se stesso, se il side-effect non c’è, oppure il nuovo valore che dovrà assumere.

```
TRANS
 case
   PC = 1 : next(PC) = 2 & next(result)  =0 & next(i) = i & next(vett[0])
= vett[0] & next(vett[1]) = vett[1] & next(vett[2]) = vett[2] ;
   PC = 2 : next(PC) = 3 & next(result) = result & next(i)  =0 &
next(vett[0]) = vett[0] & next(vett[1]) = vett[1] & next(vett[2]) =
vett[2] ;
   PC = 3 &  i<3 : next(PC) = 4 & next(result) = result & next(i) = i &
next(vett[0]) = vett[0] & next(vett[1]) = vett[1] & next(vett[2]) =
vett[2] ;
   PC = 3 & !( i<3) : next(PC) = 9 & next(result) = result & next(i) = i
& next(vett[0]) = vett[0] & next(vett[1]) = vett[1] & next(vett[2]) =
vett[2] ;
   PC = 4 &  vett[0]>result & i=0 : next(PC) = 5 & next(result) = result
& next(i) = i & next(vett[0]) = vett[0] & next(vett[1]) = vett[1] &
next(vett[2]) = vett[2] ;
   PC = 4 &  vett[1]>result & i=1 : next(PC) = 5 & next(result) = result
& next(i) = i & next(vett[0]) = vett[0] & next(vett[1]) = vett[1] &
next(vett[2]) = vett[2] ;
   PC = 4 &  vett[2]>result & i=2 : next(PC) = 5 & next(result) = result
& next(i) = i & next(vett[0]) = vett[0] & next(vett[1]) = vett[1] &
next(vett[2]) = vett[2] ;
   PC = 4 & !( vett[0]>result) & i=0 : next(PC) = 6 & next(result) =
result & next(i) = i & next(vett[0]) = vett[0] & next(vett[1]) = vett[1]
& next(vett[2]) = vett[2] ;
   PC = 4 & !( vett[1]>result) & i=1 : next(PC) = 6 & next(result) =
result & next(i) = i & next(vett[0]) = vett[0] & next(vett[1]) = vett[1]
& next(vett[2]) = vett[2] ;
   PC = 4 & !( vett[2]>result) & i=2 : next(PC) = 6 & next(result) =
result & next(i) = i & next(vett[0]) = vett[0] & next(vett[1]) = vett[1]
& next(vett[2]) = vett[2] ;
   PC = 5 & i=0 : next(PC) = 6 & next(result)  = vett[0] & next(i) = i &
next(vett[0]) = vett[0] & next(vett[1]) = vett[1] & next(vett[2]) =
vett[2] ;
   PC = 5 & i=1 : next(PC) = 6 & next(result)  = vett[1] & next(i) = i &
next(vett[0]) = vett[0] & next(vett[1]) = vett[1] & next(vett[2]) =
vett[2] ;
   PC = 5 & i=2 : next(PC) = 6 & next(result)  = vett[2] & next(i) = i &
next(vett[0]) = vett[0] & next(vett[1]) = vett[1] & next(vett[2]) =
vett[2] ;
   PC = 6 : next(PC) = 7 & next(result) = result & next(i) = i &
next(vett[0]) = vett[0] & next(vett[1]) = vett[1] & next(vett[2]) =
vett[2] ;
   PC = 7 : next(PC) = 8 & next(result) = result & next(i) = i+1 &
next(vett[0]) = vett[0] & next(vett[1]) = vett[1] & next(vett[2]) =
vett[2] ;
   PC = 8 : next(PC) = 3 & next(result) = result & next(i) = i &
next(vett[0]) = vett[0] & next(vett[1]) = vett[1] & next(vett[2]) =
```

```
vett[2] ;
   PC = 9 : next(PC) = 9 & next(result) = result & next(i) = i &
next(vett[0]) = vett[0] & next(vett[1]) = vett[1] & next(vett[2]) =
vett[2] ;
   1: next(PC)=PC & next(vett[0]) = vett[0] & next(vett[1]) = vett[1] &
next(vett[2]) = vett[2] & next(result) = result & next(i) = i ;
esac
```

Inoltre, considerando che NuSMV è in grado di gestire i vettori solamente se i loro indici sono costanti numeriche, abbiamo dovuto trovare un modo per gestire casi particolari come vettori i cui indici sono costituiti da espressioni (per esempio vett[i + j]). In questo caso, il procedimento seguito è stato di prendere ogni variabile riferita nel vettore, generare tutte le possibili combinazioni ottenute valorizzando le variabili all’interno del loro range e infine valutare l’espressione matematica ottenuta, ricavandone il risultato. Il calcolo dell’espressione algebrica risultante è stato affidato ad una libreria esterna, Eval, la quale effettua calcoli matematici su ogni tipo di espressione riguardante gli interi. Continuando l’esempio, considerando i e j con valori 0 o 1, le loro possibili combinazioni restituiscono:

```
vett[0 + 1] = vett[1]
vett[1 + 0] = vett[1]
vett[0 + 0] = vett[0]
vett[1 + 1] = vett[2]
```

Il vettore così ottenuto viene utilizzato all’interno dei case nella sezione TRANS, consentendo una corretta interpretazione da parte di NuSMV.

Infine viene codificata la parte di smv relativa al modulo MAIN, il cui compito sarà quello di creare un'istanza del modulo precedentemente realizzato e tradotto, inizializzando il suo valore del PC a 1.

```
MODULE main
VAR
 m : Massimo_massimo;

ASSIGN
 init(m.PC):=1;
```

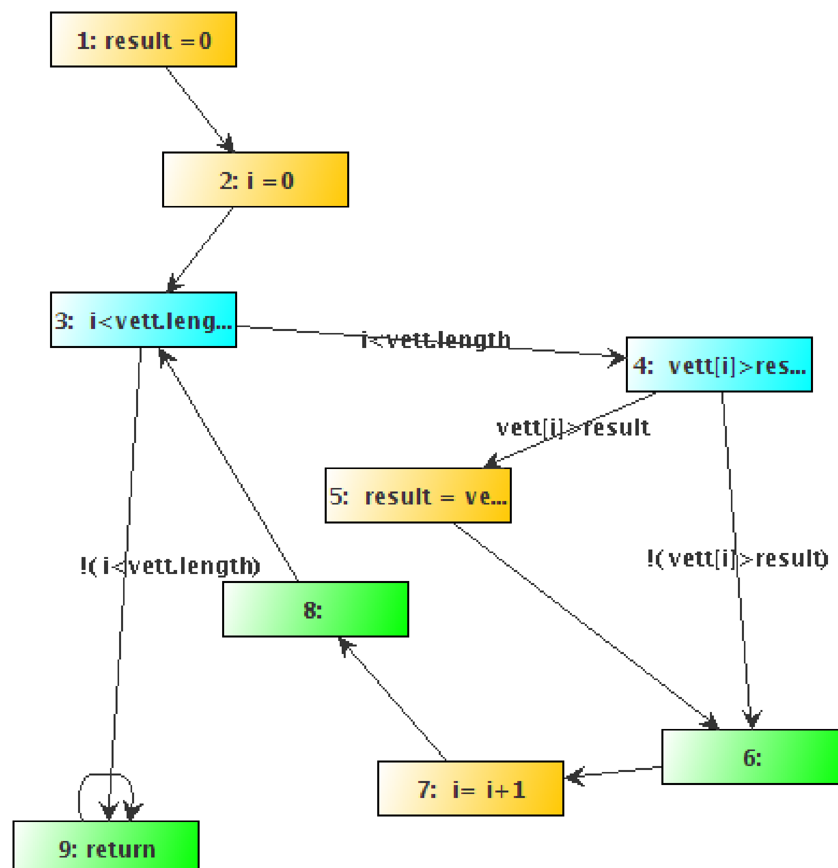
La sezione LTLSPEC contiene (commentati) dei template delle più comuni valutazioni fattibili sul codice smv realizzato (terminazione, correttezza parziale e correttezza totale). Inoltre qui verrà creata la condizione relativa all'imposizione del cammino scelto dall'utente, di cui si parlerà diffusamente nel capitolo successivo.

```
LTLSPEC
--TERMINAZIONE
--m.PRE -> F(m.TERM);
--CORRETTEZZA PARZIALE
--m.PRE -> G(m.TERM ->m.POST);
--CORRETTEZZA TOTALE
--m.PRE -> (F(m.TERM) & G(m.TERM ->m.POST));
--CAMMINO
```

Come già citato nel corso della relazione, la traduzione è esente dai problemi riscontrati in JavaToSmv relativi ad uguaglianze, incrementi e decrementi di variabili (i==1 viene corretto in i=1, cosi come i++ viene tradotto automaticamente in i=i+1).

Jasmine - 4. Test cammini

La fase di testing del software richiede una percentuale molto alta dello sforzo totale necessario per produrre un prodotto software ed è molto soggetta all'errore umano, riuscire ad automatizzare questa fase rappresenterebbe un traguardo molto importante nell'ingegneria del software verso l'avvicinamento alle altre scienze ingegneristiche già consolidate.
In questo contesto la presente tesina prenderà in considerazione la generazione automatica di casi di test a scatola bianca, nello specifico verrà presa in considerazione la generazione di test path-oriented, cioè che non lasciano all'elaboratore la scelta del cammino da percorrere, ma ne impongono uno.
E' l'utente stesso a imporre il cammino a partire dal grafo generato dal sorgente java. Qui di seguito è mostrato come esempio il grafo di un metodo che calcola e ritorna il massimo tra gli elementi di un vettore



Nella finestra principale l'utente può inserire i nodi del cammino utilizzando i numeri che identificano i nodi stessi e quale tipo di attraversamento testare.

Scelta cammino di test
Inserisci la lista ordinata di nodi separati da spazi e seleziona il tipo di cammino da percorrere
Attraversamento

Per facilità di implementazione e usabilità si è scelto di far inserire all'utente la sequenza di nodi piuttosto che la sequenza di archi del cammino, ma ovviamente le due rappresentazioni sono del tutto equivalenti.
Una volta che si ha a disposizione la sequenza di nodi che formano il cammino, l'output dell'applicazione, tramite l'utilizzo di NuSMV, sarà un caso di test che obbligherà il flusso di controllo dell'applicazione a passare per i nodi specificati secondo le specifiche del tipo di attraversamento scelto.
L'applicazione consente di scegliere tra tre tipi di attraversamenti:

1. attraversamento semplice
2. cammino ordinato
3. cammino completo

Attraversamento semplice

Si è deciso di lasciare completa libertà all'utente sulla scelta del cammino da percorrere, l'unica condizione è che il cammino scelto deve corrispondere ad un sottoinsieme dei nodi del grafo generato dall'applicazione, è accettata infatti qualsiasi forma di cammino incompleto o path-segment.
Il cammino viene codificato tramite una LTLSPEC che impone, date le precondizioni, il passaggio per gli archi del grafo di flusso che congiungono i nodi specificati.
Possiamo per esempio prendere in considerazione la seguente lista di nodi di input: "1 2 6 7"; verrà codificata in una LTLSPEC nella forma:

```
metodo.PRE -> G( metodo.PC = 1 -> X metodo.PC != 2) | G( metodo.PC = 2 ->
X metodo.PC != 6) | G( metodo.PC = 6 -> X metodo.PC != 7);
```

Intuitivamente stiamo dicendo che la verifica delle precondizioni implica che il la variabile PC non passerà mai da 1 a 2 oppure non passerà mai da 2 a 6 oppure non passerà mai da 6 a 7. Questa LTLSPEC viene inserita nel file smv generato automaticamente insieme al grafo di flusso, dato il sorgente java del metodo.
Per ottenere il caso di test per il cammino "1 2 6 7" dobbiamo far trovare a NuSMV un contro esempio alla specifica che abbiamo dato; trovare un controesempio vorrà dire trovare uno stato iniziale che, date le precondizioni, rende falsi tutti i connettivi G utilizzati, dicendo di fatto che la variabile PC passa **almeno una volta** da 1 a 2, da 2 a 6 e da 6 a 7, cioè il cammino da noi imposto.
Quindi per una lista ordinata di n nodi, questo tipo di attraversamento trova un caso di test per il quale, per ogni coppia di nodi consecutivi a_i $a_{i+1,}$ il flusso di controllo passa almeno una volta da a_i a_{i+1}.

Cammino ordinato

In questo caso si impone una condizione di ordinamento più stringente, prendendo in considerazione l'esempio precedente, viene codificata una LTLSPEC del tipo:

```
m.PRE -> (G !(m.PC=1) | G !(m.PC=2) | G !(m.PC=6) | G !(m.PC=7)) | (G
!((m.PC!=2 U m.PC=1) & (m.PC!=6 U m.PC=2) &(m.PC!=7 U m.PC=6)))
```

Intuitivamente si sta dicendo che, date le precondizioni, possono accadere due cose: o il flusso di controllo non toccherà mai il nodo 1 o non toccherà mai il 2 e cosi via, oppure non sarà mai vero che passa da un nodo diverso da 2 a 1 e che passa da un nodo diverso da 6 a 2 ecc.
L'eventuale controesempio trovato quindi prevede che data una lista di n nodi, ogni nodo a_i deve essere toccato, ma non può esserlo prima che sia toccato a_{i-i}.
Nel caso in cui cui sono presenti più nodi uguali nella lista, la seconda serie di condizioni non conterrà le occorrenze del nodo successive alla prima: prendiamo in considerazione l'esempio "1 2 6 7 2 8"; la LTLSPEC generata è la seguente:

```
m.PRE -> (G !(m.PC=1) | G !(m.PC=2) | G !(m.PC=6) | G !(m.PC=7) | G
!(m.PC=8)) | (G  !((m.PC!=2 U m.PC=1) & (m.PC!=6 U m.PC=2) &(m.PC!=7 U
m.PC=6) &(m.PC!=8 U m.PC=2)))
```

E' importante notare che questa LTLSPEC consente l'esistenza di cicli all'interno del cammino.

Cammino completo

Il terzo ed ultimo tipo di attraversamento è quello che impone l'attraversamento della lista di nodi nell'esatto ordine in cui è inserita e senza alcuna ripetizione. La LTLSPEC è del tipo:

m.PRE -> G !(m.PC=1 & X m.PC=2 & X X m.PC=6 & X X X m.PC=7)

In questo caso il controesempio trovato attraversa i nodi nell'esatto ordine inserito e non consente cicli all'interno del cammino a meno che non siano esplicitamente indicati ripetendo la sequenza di nodi del ciclo stesso tante volte quante iterazioni si desiderano.

Per rendere possibile la ricerca di casi di test è necessario utilizzare l'eseguibile NuSMV con l'opzione *-bmc*, cioè utilizzare il Bounded Model Checking sulla LTLSPEC. Abbinata a questa opzione è possibile specificare la lunghezza del bounded model checking, cioè la profondità nell'albero degli stati possibili fina alla quale NuSMV cercherà un contro esempio.
NuSMV cercherà quindi una sequenza di stati lunga al più quanto specificato nell'opzione *-bmc_length* e che renda falsa la LTLSPEC, se la sequenza di stati del contro esempio è più lunga del limite imposto dall'opzione, non verrà trovato alcun contro esempio.
E' chiaro allora che se NuSMV trova il contro esempio esso vale sicuramente come caso di test, se non lo trova non è detto che il caso di test non esista, è possibile o che debba essere aumentata la lunghezza del model checking o che il cammino indicato non è effettivamente percorribile.

Per illustrare al meglio la ricerca di un caso di test è possibile far riferimento al grafo precedente. Supponiamo di scegliere il cammino "1 2 3 9" con tipo di attraversamento semplice, dicendo intuitivamente che vogliamo che il programma termini, non specificando il passaggio o meno

nel ciclo. Inserendo la stringa precedente nell'apposito campo di testo e premendo il pulsante "Testa un cammino", il file smv viene aggiornato aggiungendo la riga

```
m.PRE -> G(m.PC = 1 -> X m.PC != 2) | G(m.PC = 2 -> X m.PC != 3) | G(m.PC = 3 -> X m.PC != 4)
```

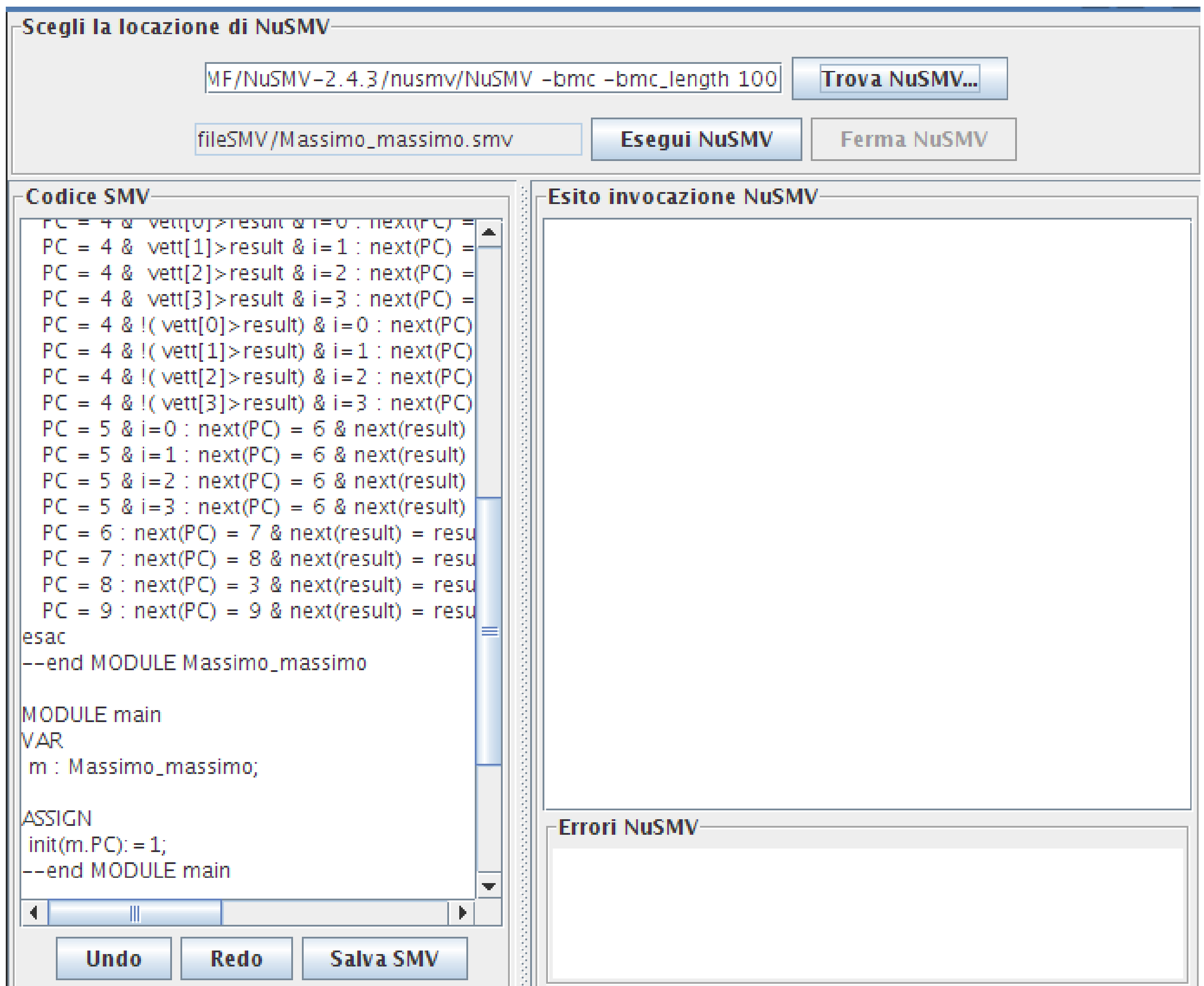
Per comodità si riporta la sezione VAR del file generato:

```
VAR
 result : -3..3;
 i : 0..4;
 vett : array 0..3 of -3..3;
 PC : 1..9;
```

e le precondizioni:

```
PRE := vett[0] >= result & vett[1] >= result & vett[2] >= result & vett[3] >= result & i = 0;
```

Premendo il pulsante "Esegui NuSMV" della finestra principale si aprirà la finestra dedicata all'esecuzione di NuSMV che avrà come input il file smv autogenerato:



L'utente dovrà solo indicare il path dell'eseguibile ed eventualmente modificare l'opzione -bmc_length. A questo punto premendo "Esegui NuSMV" partirà la ricerca del contro esempio alla nostra specifica:

fileSMV/Massimo_massimo.smv Esegui NuSMV Ferma NuSMV

Esito invocazione NuSMV

```
-- no counterexample found with bound 19
-- no counterexample found with bound 20
-- no counterexample found with bound 21
-- no counterexample found with bound 22
-- specification (m.PRE -> (( G (m.PC = 1 -> X m.PC != 2) | G (m.PC = 2 -> X m.PC != 3)) | G (m.PC = 3 -> X m.PC != 9))) is false
-- as demonstrated by the following execution sequence
Trace Description: BMC Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
 m.result = 0
 m.i = 4
 m.vett[0] = 0
 m.vett[1] = 0
 m.vett[2] = 0
 m.vett[3] = 0
 m.PC = 1
 m.PRE = 1
 m.TERM = 0
-> Input: 1.2 <-
-> State: 1.2 <-
 m.PC = 2
-> Input: 1.3 <-
-> State: 1.3 <-
 m.i = 0
 m.PC = 3
-> Input: 1.4 <-
-> State: 1.4 <-
 m.PC = 4
-> Input: 1.5 <-
-> State: 1.5 <-
 m.PC = 6
-> Input: 1.6 <-
```

Come possiamo vedere è stato trovato un contro esempio tentando con un limite di 24 stati, se fosse stato lasciato il limite di default a 10 non sarebbe stato in grado di trovare alcun contro esempio. Per comodità si riporta di seguito il contro esempio completo:

```
-> State: 1.1 <-
  m.result = 0
  m.i = 4
  m.vett[0] = 0
  m.vett[1] = 0
  m.vett[2] = 0
  m.vett[3] = 0
  m.PC = 1
  m.PRE = 1
  m.TERM = 0
-> Input: 1.2 <-
-> State: 1.2 <-
  m.PC = 2
-> Input: 1.3 <-
-> State: 1.3 <-
```

```
  m.i = 0
  m.PC = 3
-> Input: 1.4 <-
-> State: 1.4 <-
  m.PC = 4
-> Input: 1.5 <-
-> State: 1.5 <-
  m.PC = 6
-> Input: 1.6 <-
-> State: 1.6 <-
  m.PC = 7
-> Input: 1.7 <-
-> State: 1.7 <-
  m.i = 1
  m.PC = 8
-> Input: 1.8 <-
-> State: 1.8 <-
  m.PC = 3
-> Input: 1.9 <-
-> State: 1.9 <-
  m.PC = 4
-> Input: 1.10 <-
-> State: 1.10 <-
  m.PC = 6
-> Input: 1.11 <-
-> State: 1.11 <-
  m.PC = 7
-> Input: 1.12 <-
-> State: 1.12 <-
  m.i = 2
  m.PC = 8
-> Input: 1.13 <-
-> State: 1.13 <-
  m.PC = 3
-> Input: 1.14 <-
-> State: 1.14 <-
  m.PC = 4
-> Input: 1.15 <-
-> State: 1.15 <-
  m.PC = 6
-> Input: 1.16 <-
-> State: 1.16 <-
  m.PC = 7
-> Input: 1.17 <-
-> State: 1.17 <-
  m.i = 3
  m.PC = 8
-> Input: 1.18 <-
```

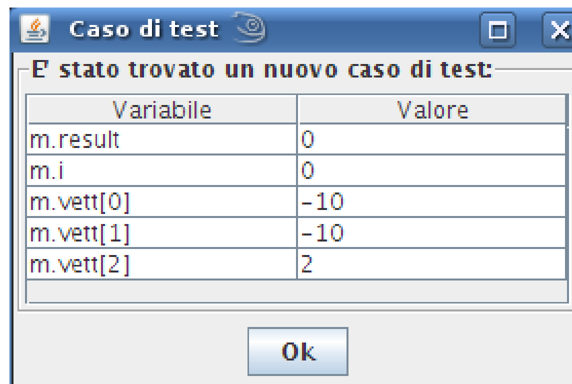
```
-> State: 1.18 <-
  m.PC = 3
-> Input: 1.19 <-
-> State: 1.19 <-
  m.PC = 4
-> Input: 1.20 <-
-> State: 1.20 <-
  m.PC = 6
-> Input: 1.21 <-
-> State: 1.21 <-
  m.PC = 7
-> Input: 1.22 <-
-> State: 1.22 <-
  m.i = 4
  m.PC = 8
-> Input: 1.23 <-
-> State: 1.23 <-
  m.PC = 3
-> Input: 1.24 <-
-> State: 1.24 <-
  m.PC = 9
  m.TERM = 1
```

Per arrivare al nodo con PC = 9 e quindi, come anche indicato nello stato 1.24 dell'output, a raggiungere le condizioni di terminazione, è stato necessario ciclare per 4 volte fino a raggiungere la condizione i<vett.length; in ogni caso abbiamo raggiunto il nostro scopo, infatti il caso di test

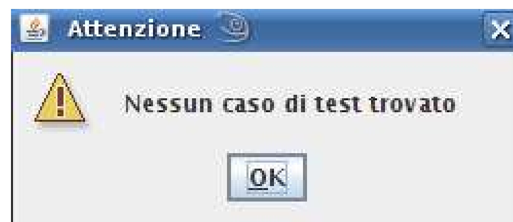
```
 m.vett[0] = 0
 m.vett[1] = 0
 m.vett[2] = 0
 m.vett[3] = 0
```

fa passare il flusso di controllo per i nodi 1 2 3 9.

L'applicazione è stata integrata con la classe di parsing della tesina JTrafficControl, in modo da presentare all'utente i risultati dell'esecuzione di NuSMV in maniera più chiar; se viene trovato un controesempio comparirà una finestra pop up con le variabili e i rispettivi valori:



In caso di controesempio non trovato comparirà un messaggio di errore:



Jasmine - 5. Integrazione con OOPS

La tesi OOPS (Object Oriented Project Support) descrive un'applicazione per il supporto allo sviluppo di software object oriented in tutte le fasi della sua progettazione. E' posta particolare enfasi sulla separazione tra fase di analisi e fase di specifica e realizzazione, e su come i vari tool UML oggi presenti sul mercato non riescano ad accompagnare lo sviluppatore in maniera adeguata durante tutte queste fasi.
L'applicazione in particolare si integra con UMLgc (UML code generator) prodotto della tesi di Ottaviani, per raggiungere lo scopo di arrivare fino alla fase di generazione del codice.
Siamo in particolare interessati quindi alle classe UMLClass che dispone della funzione

```
public void generateCode(PrintWriter out)
```

Se questa funzione viene chiamata, viene generato il codice della classe inclusi quindi tutti i suoi metodi e scritto su `out`.

Per evitare di legarsi troppo alle classi di UMLgc e della tesi OOPS, è stato deciso di implementare un'interfaccia che eventuali applicazioni esterne devono implementare per utilizzare le funzionalità di Jasmine.

```
package it.uniroma1.dis.jasmine;

import java.io.PrintWriter;

public interface ICodeGenerator {

   public void generateCode(PrintWriter out);

   public String getClassName();
}
```

La funzione `getClassName` è necessaria per creare un file il cui nome corrisponda alla classe generata dalla funzione `generateCode`.

E' stato creato inoltre un wrapper che lega ICodeGenerator alle API di Jasmine per la generazione del grafo di flusso e del codice smv relativo al sorgente java. Per utilizzare le suddette API è necessario conoscere il path del file java che contiene il sorgente, per cui è stato sufficiente creare una funzione che, dato ICodeGenerator, scrivesse su file e ritornasse il path del file scritto:

```
package it.uniroma1.dis.jasmine;

import java.io.File;
import java.io.IOException;
import java.io.PrintWriter;

public class CodeGeneratorWrapper {

  public static String writeJavaFileFromCodeGenerator(ICodeGenerator cg)
{

    String className = cg.getClassName();

                          String          path          =
System.getProperty("java.io.tmpdir")+"/"+className+".java";

    try {
      File f = new File(path);
      PrintWriter pw = new PrintWriter(f);
      cg.generateCode(pw);
    } catch (IOException ioe) {
      return null;
    }

    return path;
  }

}
```

Il codice generato viene scritto su un file nella cartella temporanea di sistema ed il suo path viene ritornato se la scrittura va a buon fine, altrimenti ritorna `null`.

Jasmine - 6. Interfaccia grafica

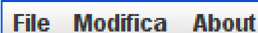
• 6.1. Panoramica libreria JGraph

Uno dei passi eseguiti da Jasmine prima di invocare nuSMV per ottenere i casi di test è generare il grafo di flusso di un metodo e far scegliere il percorso da testare. Abbiamo quindi deciso di offrire una visualizzazione grafica del grafo di flusso, onde facilitare la comprensione e quindi la scelta del cammino da testare. Per disegnare il grafo abbiamo utilizzato la parte open-source della libreria JGraph.
JGraph offre la possibilità di disegnare e interagire con grafi, utilizzando ed estendendo le API Swing di Java. La versione open-source contiene solamente le funzionalità base, che consistono nel disegnare un insieme di nodi collegati da archi, lasciando a cura dello sviluppatore il loro posizionamento; JGraph offre, inoltre, delle librerie aggiuntive che automatizzano il layout di nodi ed archi, ma queste librerie sono disponibili solamente con licenza commerciale.
Per poter sfruttare le potenzialità di JGraph abbiamo dovuto effettuare un'operazione di conversione, dalla rappresentazione interna del grafo di flusso alla rappresentazione della libreria grafica. Questa conversione è realizzata all'interno del metodo "createGraphFromList" della classe "it.uniroma1.dis.jasmine.ui.GraphViewer": l'oggetto restituito dal metodo, una lista di oggetti "org.jgraph.graph.DefaultGraphCell" è pronto per essere disegnato come grafo. E' interessante notare come in JGraph l'oggetto che rappresenta un'arco, ossia "org.jgraph.graph.DefaultEdge", estenda in realtà l'oggetto cella "DefaultGraphCell".
Per ovviare al problema dell'ordinamento di nodi e archi del grafo, visto che le librerie open-source non offrono alcun layout automatizzato, abbiamo dovuto inventare un modo per posizionare i nodi in modo da rendere il grafo il più possibile leggibile.

6.2. Struttura dell'interfaccia grafica

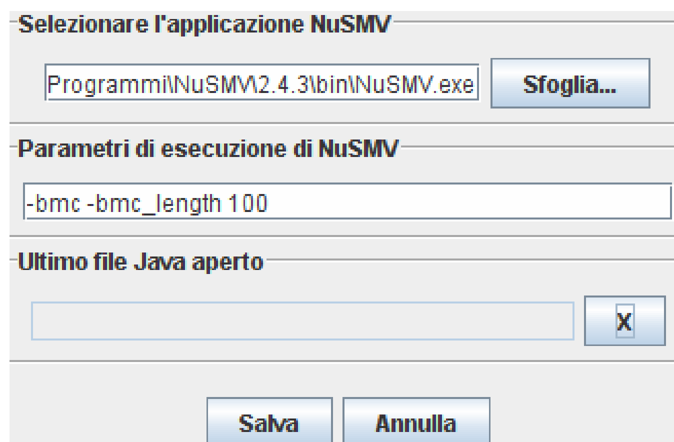
L'interfaccia grafica di Jasmine è stata sviluppata con lo scopo di raggruppare in un'unica applicazione le diverse operazioni che portano da un file sorgente java ad un file smv, alla sua modifica, e al suo utilizzo con l'eseguibile di NuSMV; il tutto senza aprire altre applicazioni o console di testo.

E' possibile interagire con l'applicazione tramite un menu in alto



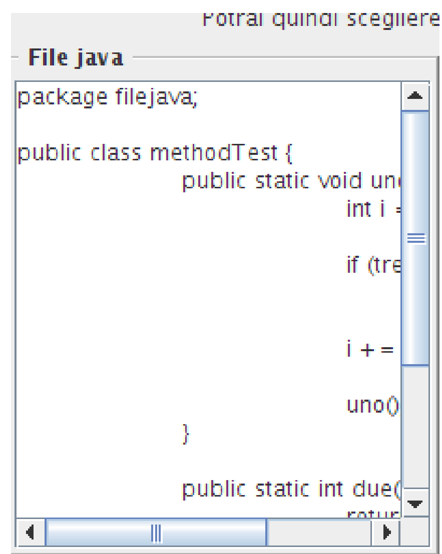
e una barra dei pulsanti in basso

Il menu principale ha una voce "About" che riassume funzionalità e autori dell'applicazione. E' inoltre raggiungibile, a partire dal menu "Modifica", una schermata che consente di impostare le opzioni di Jasmine, tra cui la posizione dell'eseguibile NuSMV (qualora non si trovasse all'interno del path del sistema), le opzioni di esecuzione di NuSMV e l'ultimo file Java caricato. Queste impostazioni vengono salvate su file e caricate ad ogni avvio.



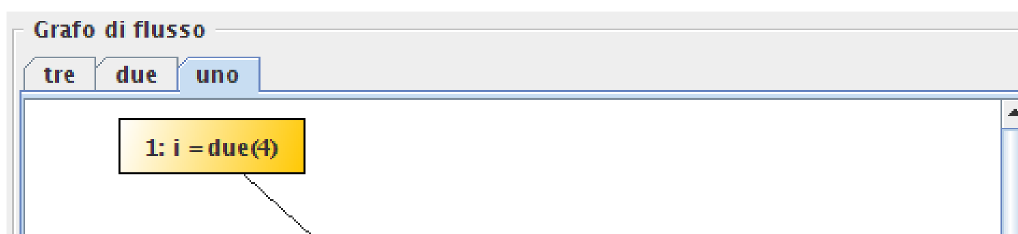
Il punto di partenza nell'utilizzo dell'applicazione è l'apertura di un file sorgente java, funzionalità raggiungibile dal menu principale alla voce "File". Una volta aperto il file sorgente comparirà il suo contenuto

nell'apposita area di testo denominata "File Java".



A questo punto è possibile cominciare ad utilizzare le funzionalità dell'applicazione. Tipicamente l'utente vorrà creare il grafo di flusso del codice sorgente. E' sufficiente premere il pulsante "Crea Grafo" per far comparire il grafo di flusso del codice nell'area di testo denominata "Grafo di flusso". I nodi sono identificati da un numero che corrisponderà alla variabile PC nel file smv e dall'eventuale riga di codice java; gli archi del grafo sono dove necessario etichettati con la condizione da soddisfare perché il flusso logico percorra quell'arco. Nel caso in cui i nodi/archi fossero sovrapposti e quindi non leggibili è possibile spostarli tramite drag'n'drop del mouse.

Nel caso in cui il file sorgente java contiene un metodo che chiama altri metodi anch'essi presenti sullo stesso file, verranno generati più grafi uno per ogni metodo. Vengono creati di conseguenza più tab, uno per ogni metodo:

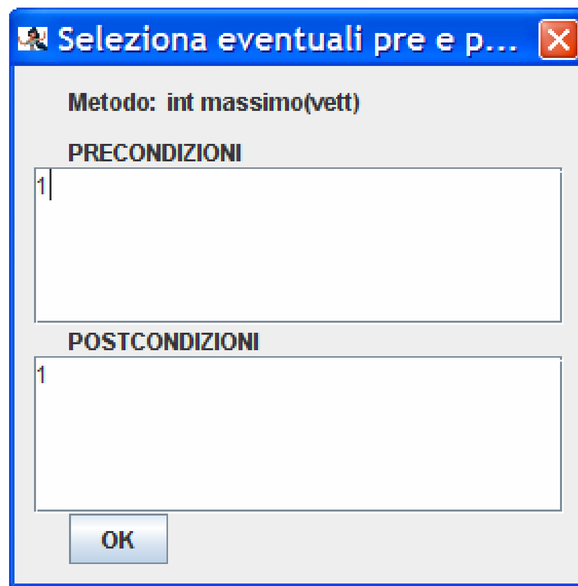


Per comodità inoltre è possibile navigare tra i diversi tab, oltre che nella modalità standard selezionando il tab stesso, anche selezionando un nodo nel grafo che ha un riferimento al metodo desiderato con il tasto destro del mouse. Prendendo come riferimento l'immagine precedente, un click con il tasto destro del mouse sul nodo identificato da "1: i = due(4)", apre il tab che contiene il grafo di flusso del metodo 2. Ogni volta che il tab mostra un diverso grafo, la sezione codice SMV a sinistra mostra il rispettivo codice SMV.

E' stata inserita inoltre un tool per esportare il grafo visualizzato come immagine PNG, è sufficiente premere il pulsante "Esporta PNG" e verrà chiesto dove salvare il file.

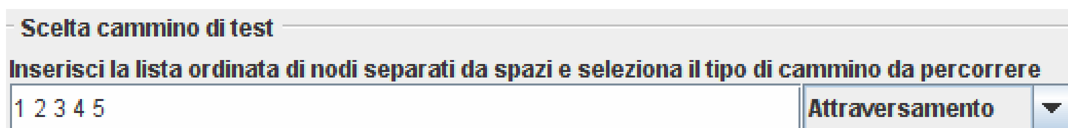
Insieme alla generazione del grafo di flusso avviene anche la generazione del codice smv, anch'esso comparirà nella sua area di testo denominata "Codice SMV". Per perfezionare il codice smv auto generato è possibile specificare pre e post condizioni e il range del dominio delle variabili premendo il tasto "Cambia opzioni". Si apriranno a questo punto delle popup in sequenza, per il metodo associato al grafo selezionato, per inserire i range delle variabili presenti in esse:

Si aprirà inoltre un’ultima popup per l'inserimento delle pre e post condizioni, che dovranno essere inserite rispettando la sintassi smv e ovviamente utilizzando gli stessi identificatori per le variabili utilizzati nel file smv. Viene mostrato il metodo con la sua segnatura e delle pre e post condizioni di default:



Prima di passare all'analisi con il tool NuSMV di cui si parla di seguito, è anche possibile, come spiegato anche nel capitolo dedicato al test sui cammini, inserire un cammino identificato da una sequenza di nodi e codificarlo in una LTLSPEC. Il cammino viene inserito specificando i nodi da percorrere, come numeri separati da spazi. E' possibile scegliere il tipo di cammino, uno tra:

- "Attraversamento": attraversamento semplice dei nodi;
- "Cammino ordinato": attraversamento dei nodi nell'ordine specificato, senza vincolo sul numero di cicli da percorrere;
- "Cammino completo": attraversamento dei nodi nell'ordine specificato e seguendo i cicli il numero specificato di volte; per specificare un passaggio in un ciclo è necessario inserire la sequenza dei nodi che compongono il ciclo;



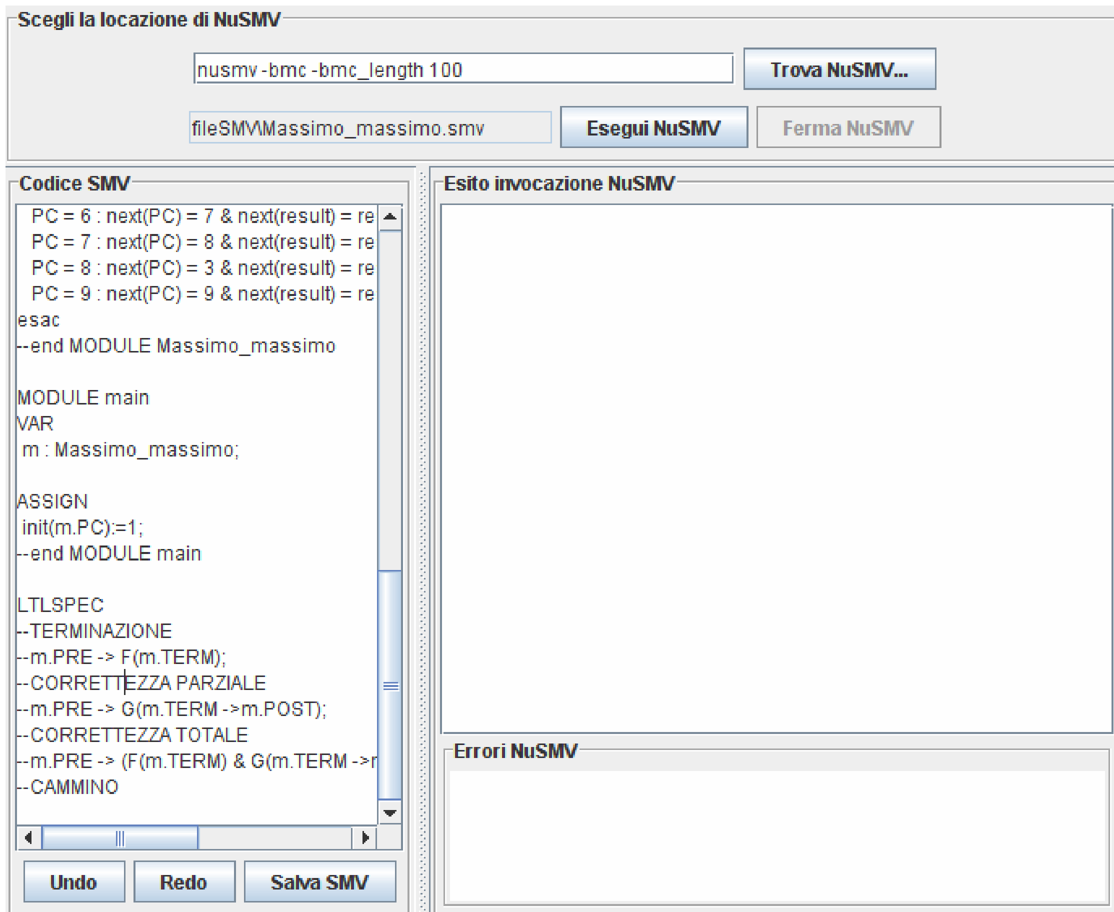
Per iniziare l'analisi è sufficiente premere il pulsante "Esegui NuSMV" per aprire la finestra che integra l'applicazione con il tool. E' importante notare che, mentre nel codice smv della finestra principale sono presenti tutti i metodi di cui sono stati anche generati i grafi, quando si entra nella modalità di analisi con NuSMV, viene preso in considerazione solo il grafo selezionato insieme alla parte del file smv di interesse.

6.3. Integrazione con nuSMV

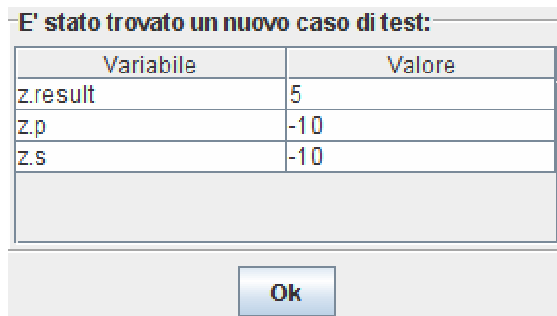
NuSMV è un tool utile per effettuare il model checking su logica temporale. Viene distribuito come eseguibile open-source, realizzato in una tecnologia non Java, per cui non facilmente integrabile in Jasmine. Abbiamo quindi optato per eseguire un'invocazione da riga di comando e in tal caso è necessario passare come parametro il nome del file smv da testare.

L'interfaccia grafica messa a disposizione consente, quindi, di scegliere la locazione dell'eseguibile nuSMV ed offre le due opzioni di esecuzione necessarie in seguito, ossia -bmc e -bmc_length, consentendo di modificarle oppure di aggiungerne di nuove. Il percorso dell'eseguibile ed i parametri di esecuzione sono presi dalle impostazioni di Jasmine, che è possibile modificare secondo le modalità descritte sopra.

Nella parte sinistra è presente il file SMV generato al passo precedente; si tratta di un semplice editor di testo, munito solamente di funzionalità di "Undo" e "Redo", ma è possibile editare il file SMV generato, a patto poi di salvare le modifiche prima di eseguire nuSMV. L'invocazione viene avviata dal pulsante "Esegui NuSMV" e, mentre è attiva, è possibile interromperla premendo l'apposito pulsante "Ferma NuSMV", qualora i tempi di esecuzione risultassero troppo lunghi. I due campi di testo "Esito invocazione" ed "Errori" mostrano l'output della console NuSMV.



Se l'esecuzione ha avuto successo, ossia NuSMV ha trovato un controesempio, tale controesempio costituisce il caso di test richiesto, che viene mostrato in una popup contenente una tabella; la tabella dei casi di test mostra le variabili iniziali ed il loro valore di test:


E' stato trovato un nuovo caso di test:

Variabile	Valore
z.result	5
z.p	-10
z.s	-10

Ok

Thread:

Per rendere possibile l’interazione tra Jasmine e NuSMV è stato necessario utilizzare dei thread separati:

- un thread principale che esegue NuSMV, per evitare azioni bloccanti sull'interfaccia e per offrire la possibilità di arrestarne l'esecuzione;
- due thread figli che leggono rispettivamente dallo standard output e dallo standard error e scrivono negli appositi campi di testo.

In questo modo Jasmine mantiene la sua struttura modulare e svincolata da applicazioni e librerie esterne in quanto è possibile cambiare versione di NuSMV e anche, potenzialmente, invocare un qualsiasi altro programma esterno.

Jasmine - 7. Conclusioni

Il lavoro di studio dietro la stesura di questa tesina ci ha fatto approfondire la nostra conoscenza delle funzionalità del tool NuSMV e in particolare il bounded model checking.
Siamo diventati inoltre maggiormente consapevoli dei limiti di questo tool nell'analisi di un linguaggio complesso come può essere quello java. A causa di questi limiti, primo fra tutti l'ingestibilità di tipi reali, i campi di applicabilità di NuSMV sono ridotti a un sottoinsieme molto ridotto dei possibili costrutti esprimibili in java. Attualmente infatti Jasmine è in grado di trovare casi di test solo per metodi java che utilizzano variabili intere o di tipo booleano.

In ogni caso con questa applicazione abbiamo cercato di gettare le basi per una possibile automatizzazione della creazione ed esecuzione dei casi di test a scatola bianca, in questo senso è stato portato avanti il lavoro di integrazione con la tesina OOPS.
Il traguardo ideale è quello di mettere a disposizione dello sviluppatore un tool che lo segue dalla fase di analisi fino alla fase di test e validazione, fase in cui l'applicazione Jasmine si inserisce.

Sviluppi futuri

I possibili sviluppi futuri su quest'applicazione possono essere relativi a miglioramenti di release future di NuSMV che diminuiscano le sue limitazioni, in questo caso potrebbe essere opportuno andare di pari passo e portare questi miglioramenti nella generazione di codice smv.
Altro possibile lavoro importante per il futuro potrebbe essere la progressiva trasformazione di Jasmine (o di un suo fork) in un plugin per l'applicazione di cui si è accennato in precedenza per completare l'opera di integrazione.

Miglioramenti

Il supporto di Jasmine per le chiamate a funzioni ha ampi margini di miglioramento, specialmente per quanto riguarda la chiamata a metodi ricorsivi con più parametri in ingresso e più chiamate a metodi all'interno di un'istruzione.

Funzionalità offerte da Jasmine:

- creazione e visualizzazione grafo di flusso di metodi Java;
- gestione di chiamate a metodi all'interno della stessa classe (mediante utilizzo di pre- e post-condizioni);
- scelta di un cammino all'interno del grafo di flusso;
- generazione di codice NuSMV associato ad una classe Java;
- generazione di casi di test per 3 tipologie di cammini (mediante l'invocazione di NuSMV).

Jasmine - 8. Bibliografia e riferimenti

Librerie e tool esterni utilizzati in Jasmine:

- Java Tree Builder (versione 1.3.2)

http://compilers.cs.ucla.edu/jtb/jtb-2003/
http://compilers.cs.ucla.edu/jtb/jtb-2003/whyvisitors/index.html

- NuSMV (versione 2.4.3)

http://nusmv.irst.itc.it/
http://nusmv.fbk.eu/NuSMV/tutorial/v24/tutorial.pdf

- JGraph (versione 5.12)

http://www.jgraph.com/jgraph.html
http://www.jgraph.com/pub/jgraphmanual.pdf

Eval – a simple expression evaluator for Java (versione 0.4)

https://eval.dev.java.net/

Materiale di supporto del corso:

- Corso di Metodi Formali

http://www.dis.uniroma1.it/~tmancini/index.php?currItem=teaching.mfis.programma

- Tesina "JavaToSMV"

Daniele Ippoliti, Marco Leone - Generazione automatica di codice nuSmv per la verifica di metodi Java

http://www.dis.uniroma1.it/~tmancini/index.php?currItem=teaching.mfis.materiale.progetti.dettaglio&idprogetto=ippoliti-leone

- Tesina "OOPS"

Michele Proni - OOPS: un’applicazione per il supporto alla progettazione e alla realizzazione di software object oriented

- Tesina "JTrafficControl"

Raffaele Giuliano, Marco Piva, Fabio Terella, Emanuele Tracanna - JTrafficControl: un'applicazione per la sintesi automatica di scheduling semaforici mediante nuSmv

http://www.dis.uniroma1.it/~tmancini/index.php?currItem=teaching.mfis.materiale.progetti.dettaglio&idprogetto=giuliano-piva-terella-tracanna