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

Indice

- Introduzione
- Creazione grafo di flusso
- Rappresentazione interna
- Conversione in SMV
- Chiamate a metodi
- Test cammini
- Esempi

Introduzione

Jasmine è un programma che effettua un parsing di una classe Java e ne crea 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 stesso di Jasmine convertire il grafo di flusso generato in codice smv.

Tale codice sarà poi 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.

Cos'è Jasmine

Creazione del grafo di flusso

- In questa fase il file Java viene utilizzato da JTB (Java Tree Builder) per ricavare la rappresentazione sintattica, sotto forma di insieme di classi automaticamente generate a partire dalla grammatica di Java, con JavaCC (Java Compiler Compiler).
- Le classi JTB che descrivono la sintassi compongono una struttura annidata che segue quella del codice Java.

Pattern Visitor

- JTB utilizza il pattern Visitor per gestire visite all'interno della struttura sintattica.
- Questo pattern comportamentale implementa due comportamenti:
  - il visitor dichiara un metodo visit() per ogni tipo di elemento da visitare;
  - l'elemento offre un metodo accept() nel quale definisce le modalità della visita;
- In JTB ogni classe di sintassi è un elemento e vengono forniti dei visitor di default, tra cui DepthFirstVisitor.

Esempio di classe di sintassi

- Una delle classi di sintassi di JTB è WhileStatement:
  - f0 -> "while"
  - f1 -> "("
  - f2 -> Expression()
  - f3 -> ")"
  - f4 -> Statement()
- f0…f4 sono nodi, rappresentati da stringhe (della grammatica) oppure altri nodi (Expression …), realizzando l'annidamento.

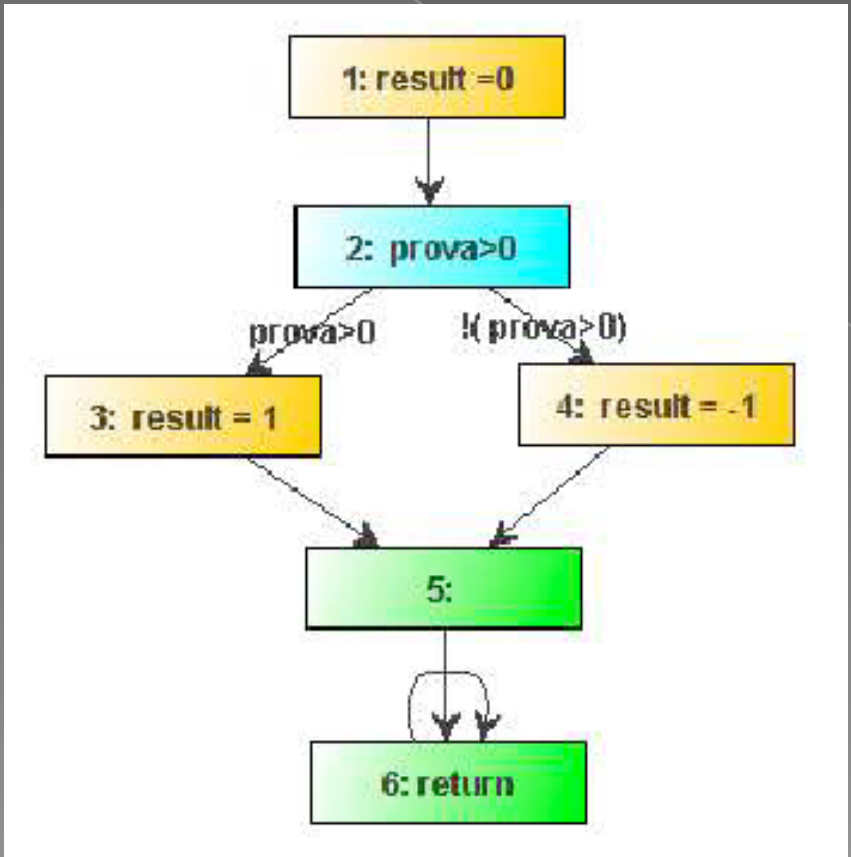
Visitor di Jasmine

- Jasmine definisce 4 visitor, che estendono DepthFirstVisitor:
  - *TranslatorVisitor*: contiene tutta la logica necessaria per estrarre il grafo di flusso dalla classe che sta visitando.
  - *ImageVisitor*: ricava una descrizione testuale di un qualunque elemento all'interno della classe
  - *LenghtVisitor*: calcola la lunghezza di un qualunque elemento (numero di istruzioni)
  - *MethodVisitor*: ricava i metodi presenti nella classe
- TranslatorVisitor crea il grafo di flusso effettuando un'unica visita, durante la quale utilizza gli altri in modo trasversale

TranslatorVisitor e annidamenti

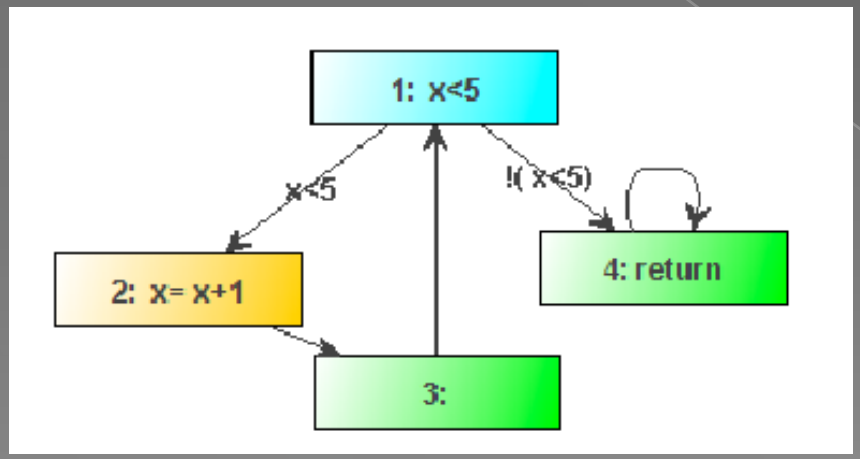
- Il TranslatorVisitor, all'inizio della visita della classe, prende i nomi di tutti i metodi in essa contenuti poi, per ogni metodo, crea un grafo di flusso e quindi effettua una visita in profondità, inserendo nel grafo corrispondente i nodi relativi alle istruzioni ed eventuali nodi senza operazione (NOP).
- L'annidamento delle istruzioni viene rispettato dalla visita in profondità sulla struttura annidata che rappresenta la sintassi della classe Java.

L'espressione if



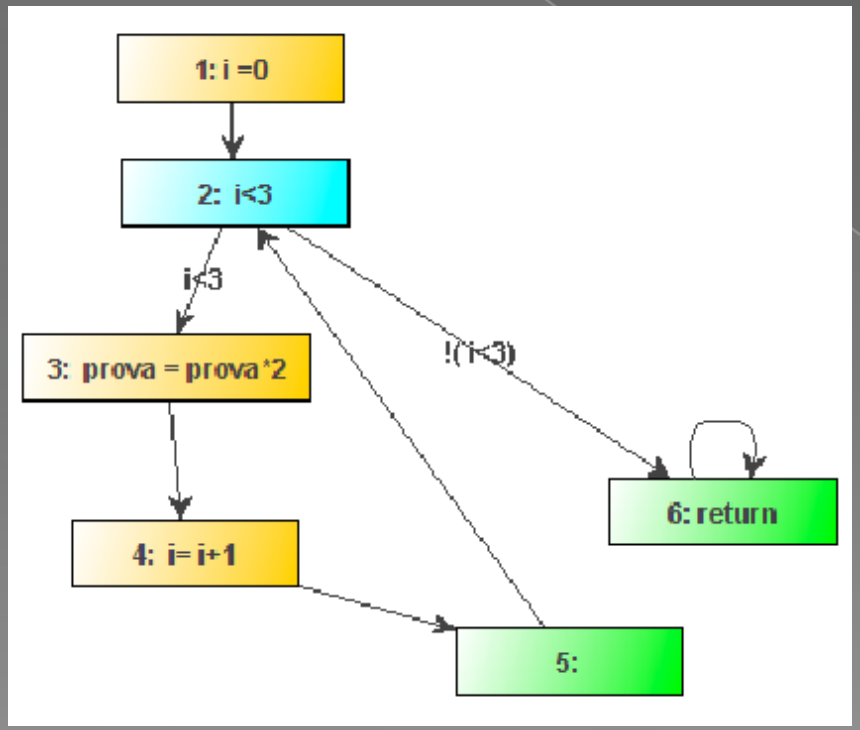

```
public class TestIf {
public static int
   testaIf(int prova) {
   int result = 0;
   if (prova > 0) {
     result = 1;
   }
   else result = -1;
   return result;
}
```

L'espressione while



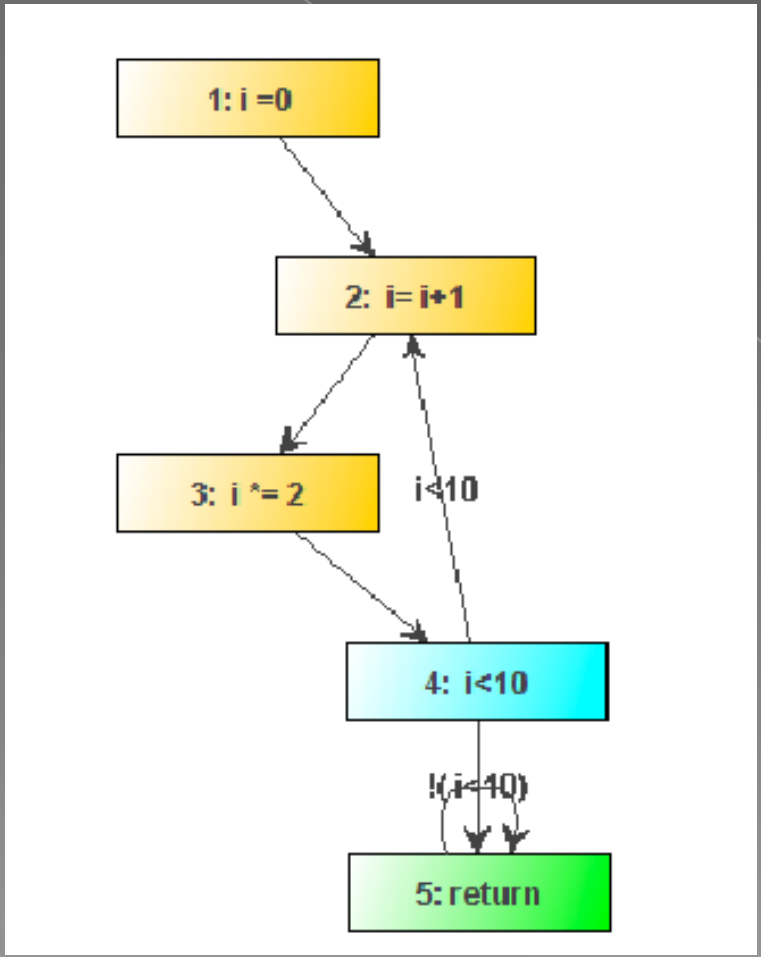

```
public class TestWhile
   {
      public static void
    testaWhile(int x) {
           while (x<5) {
                x++;
           }
      }
}
```

L'espressione for



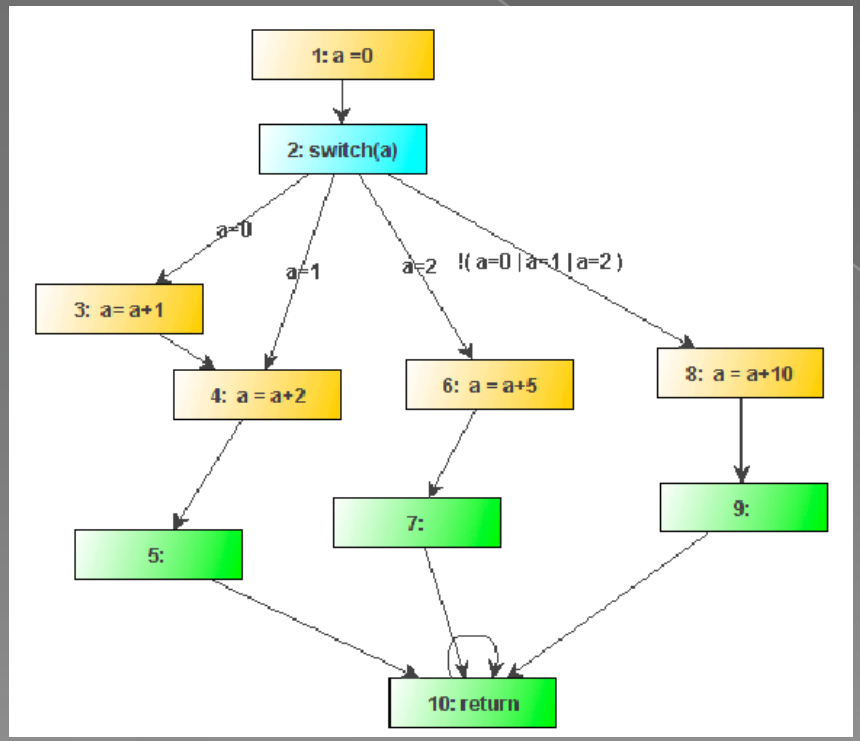

```
public class TestFor {
     public static void
   testaFor(int prova) {
for (int i = 0; i < 3; i++)
   {
      prova = prova * 2;
            }
         }
}
```

L'espressione do...while




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

L'espressione switch




```
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;
}
}
```

Rappresentazione interna

- Il grafo di flusso è costruito utilizzando tre classi:

- **GraphNode**, ogni nodo del grafo
- **Graph**, grafo di un metodo della classe
- **GraphWrapper**, classe passata al TranslatorVisitor

- Inoltre è presente la classe **MethodCall**, attributo di GraphNode e di Graph

GraphNode

- Il nodo del grafo è rappresentato dalla classe "it.uniroma1.dis.jasmine.GraphNode"; ad esso corrisponde un'istruzione del metodo Java. Introduce il concetto di Program Counter (PC), inteso come numero dell'istruzione corrente.

- Attributi:

- 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.

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.

- Attributi:

- nodes - vettore di oggetti GraphNode, uno per ogni istruzione; come detto, 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.

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.

- Attributi:

- 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.

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; se, invece, è un attributo di Graph, contiene informazioni aggiuntive riguardanti il metodo cui il grafo è associato. 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.

- Attributi:

- 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.

Conversione in smv

- Affidata alla classe FileSmvWriter.java
- Converte la struttura del grafo di flusso nel corrispondente codice smv
- Durante la visita dell'albero sintattico di JTB, le informazioni sulle variabili locali e sui parametri formali del metodo sono state memorizzate nelle opportune strutture. La stessa visita determina quante siano le istruzioni di terminazione del programma (return) con relativi valori del Program Counter, e quale sia il massimo valore del PC raggiungibile

Sezione VAR

- I range e la lunghezza degli array sono stabiliti dall'utente tramite interfaccia. Alternativamente, vengono utilizzati dei valori di default.

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

Sezione DEFINE

- 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)

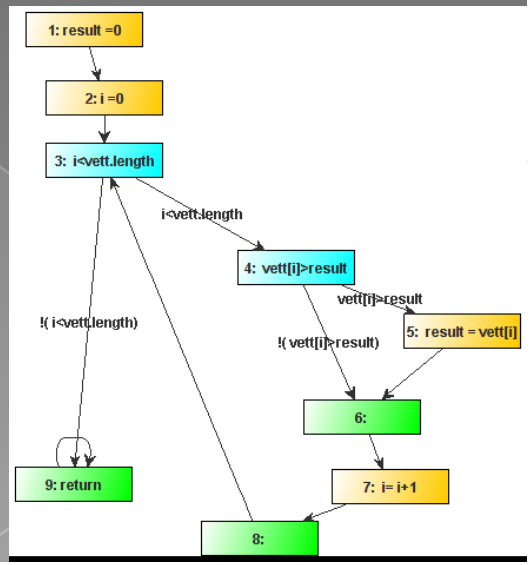
```
DEFINE
TERM := PC = 9;
PRE := 1;
POST := 1;
```

Sezione TRANS

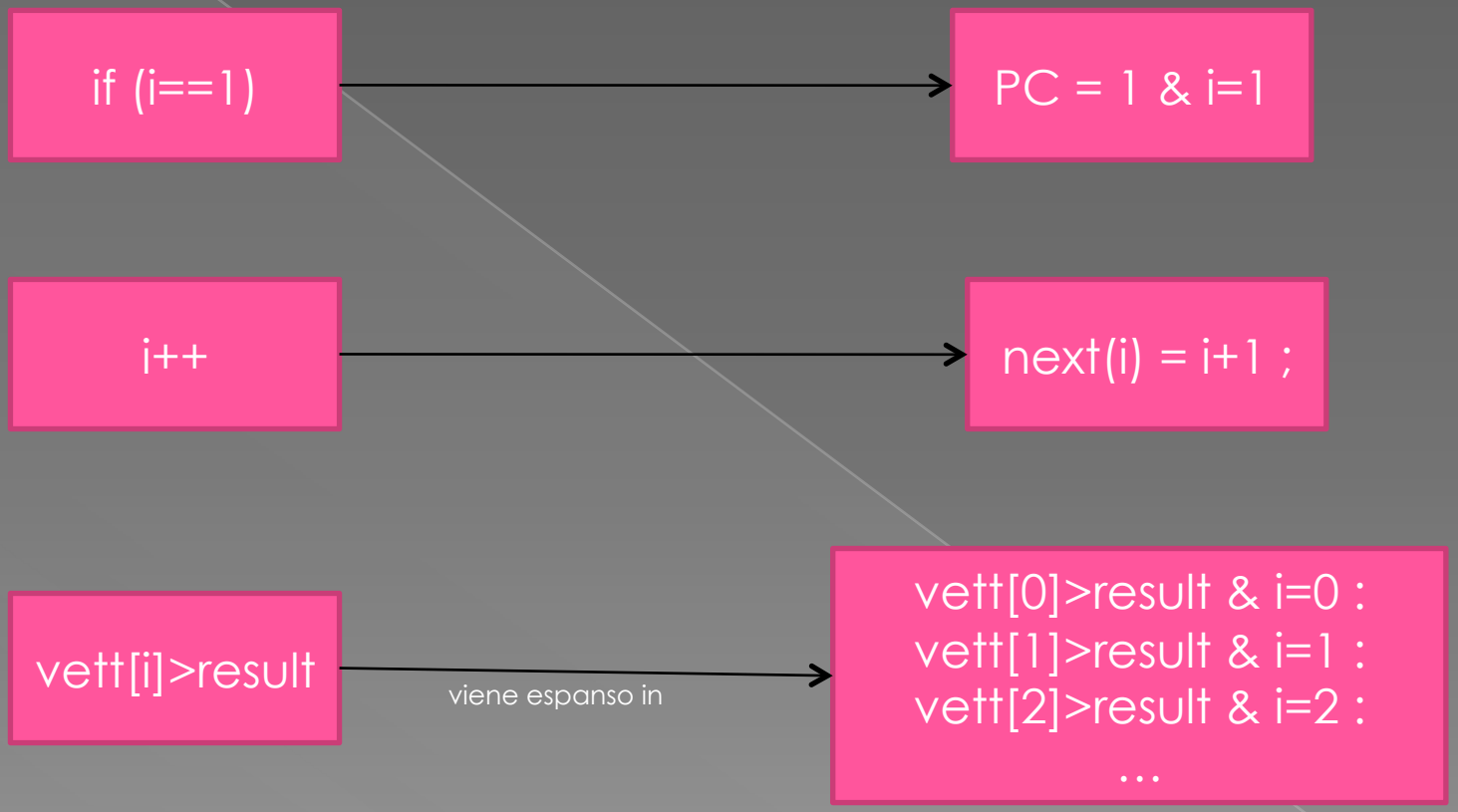
- L'oggetto Node è assegnato ad ogni riga di TRANS, e popolato durante la creazione del grafo di flusso. In esso saranno memorizzate le informazioni sul PC di partenza, il PC di destinazione, eventuali condizioni di passaggio, ed eventuale modifica ai valori delle variabili o dei parametri formali dovuta a particolari istruzionidella corrispondente riga nel programma Java di input.

Sezione TRANS: esempio

```
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
```



Sezione TRANS: correzioni



Sezione TRANS: altre correzioni

vett[i+j]>result

ANDREBBE
tradotto in...

vett[0+0]>result& i=0 & j=0 :
vett[1+0]> result & i=1 & j=0 :
vett[2+0]> result]& i=2 & j=0 :
vett[0+1]> result]& i=0 & j=1 :
vett[1+1]> result & i=1 & j=1 :
vett[2+1]> result]& i=2 & j=1 :
vett[0+2]> result & i=0 & j=2 :
vett[1+2]> result & i=1 & j=2 :
vett[2+2]> result & i=2 & j=2 :

Ma NuSMV non è in grado di gestire neanche gli indici vett[1+1]!!!! (non considera 1+1 costante)

Serve il passaggio successivo:
PRECALCOLARE OGNI ESPRESSIONE COMPOSTA NEGLI ARRAY CON LA LIBRERIA ESTERNA EVAL

if (vett[i] > vett[i-3+1-2*i+3+2*i])

vett[0]>vett[0-3+1-2*0+3+2*0] & i=0 :
vett[1]>vett[1-3+1-2*1+3+2*1] & i=1 :
vett[2]>vett[2-3+1-2*2+3+2*2] & i=2 :

EVAL

vett[0]>vett[1] & i=0 :
vett[1]>vett[2] & i=1 :
vett[2]>vett[3] & i=2 :

Sezione MAIN

- Crea un'istanza del modulo precedentemente creato, e imposta il valore iniziale del PC a 1

```
MODULE main
VAR
m : Massimo_massimo;
ASSIGN
init(m.PC):=1;
```

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 successivamente.

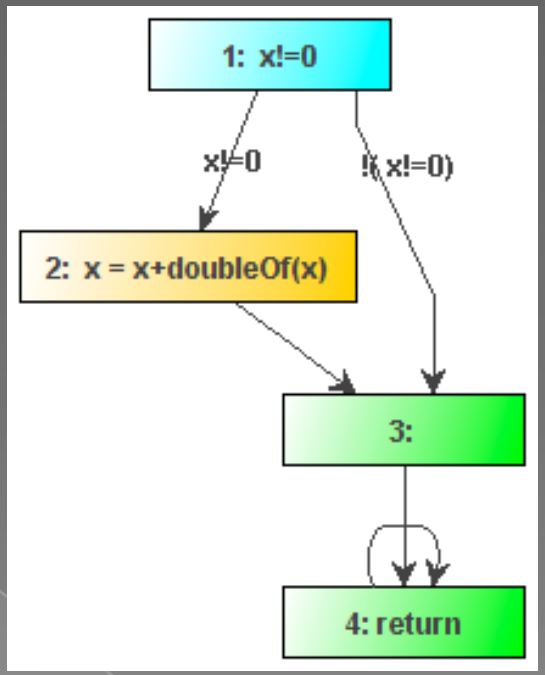
```
LTLSPEC
--TERMINAZIONE
--m.PRE -> F(m.TERM);
--CORRETTEZZA PARZIALE
--m.PRE -> G(m.TERM ->m.POST);
--CORRETTEZZA TOTALE
--m.PRE -> (F(m.TERM) & G(m.TERM ->m.POST));
--CAMMINO
```

Chiamate a metodi

- Jasmine individua chiamate a metodi all'interno della stessa classe e ne utilizza le pre e post-condizioni per sfruttarli come "stub" all'interno dei casi di test
- Internamente, il collegamento tra metodi avviene mediante l'oggetto MethodCall, presente in ogni Graph (caratterizzandolo) e in ogni GraphNode che effettua chiamata a metodo. Inoltre, questo oggetto contiene anche le pre e post-condizioni associate al metodo.

Rappresentazione grafica

- Nell'interfaccia grafica è disponibile una rappresentazione grafica del grafo associato ad ogni metodo.
- Qualora uno dei nodi dovesse effettuare una chiamata a metodo, è sufficiente cliccarci col tasto destro per selezionare il grafo del metodo chiamato.



Pre e post-condizioni

- Ogni metodo ha pre e post-condizioni che ne definiscono la semantica.
- Assunzione: se le pre-condizioni impongono condizioni sulle variabili in input e le post-condizioni impongono condizioni sulle variabili in output in funzione di quelle in input allora è possibile definire il comportamento del metodo esprimendone le condizioni in logica.

Esempio – metodi

- Consideriamo i seguenti metodi, definiti all'interno della stessa classe:

```
public int doubleOf(int y) {
   return y*2;
}
public int simpleTripleOf(int x) {
   int d = doubleOf(x);
   return x+d;
}
```

- Il metodo da testare è simpleTripleOf(x). Osserviamo che simpleTripleOf(x) utilizza doubleOf(y), passandogli un parametro.

Esempio – metodi – condizioni

- Per il medodo doubleOf(y), le condizioni sono:
  - pre: accetta numeri interi
  - post: restituisce il doppio del suo input
- Scriviamo queste condizioni nella sezione DEFINE del codice SMV per il metodo simpleTripleOf(x):

  doubleOfPRE := x>0;

  doubleOfPOST := x*2;
- Osserviamo che il parametro x è quello passato al metodo e non quello dichiarato

Esempio – metodi – SMV

- La sezione TRANS del codice SMV per il metodo simpleTripleOf(x) simula la chiamata a metodo:

```
PC = 2 & doubleOfPRE : next(PC) = 3 & next(x)  =   x
    +doubleOfPOST ;
1: next(PC)=PC & next(x) = x ;
```

- La post-condizione è considerata come se fosse il valore di ritorno del metodo chiamato.
- La pre-condizione deve essere verificata.
- La condizione di default copre il caso in cui la pre-condizione non è verificata.

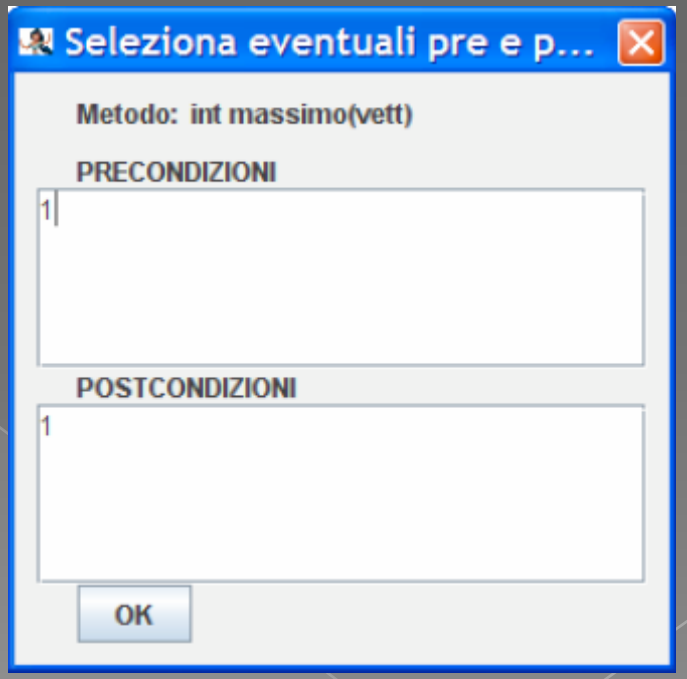
Esempio – metodi – risultato

- Osservando il risultato dell'invocazione NuSMV possiamo verificare che, caso per caso, doubleOfPOST e doubleOfPRE assumono valori in funzione del "parametro" x:

```
-> State: 1.1 <-
  t.x = 1
  t.PC = 1
  t.doubleOfPOST = 2
  t.doubleOfPRE = 1
```

Inserimento pre e post-condizioni

- Le pre e post-condizioni possono essere inserite da interfaccia grafica, cambiando le opzioni del grafo.
- JTB, durante il parsing, toglie i commenti da codice, rendendo impossibile l'inserimento delle condizioni all'interno dei commenti.



Test Cammini

Jasmine consente di definire un cammino all'interno del grafo e cercare un caso di test che lo attraversi grazie a NuSMV.

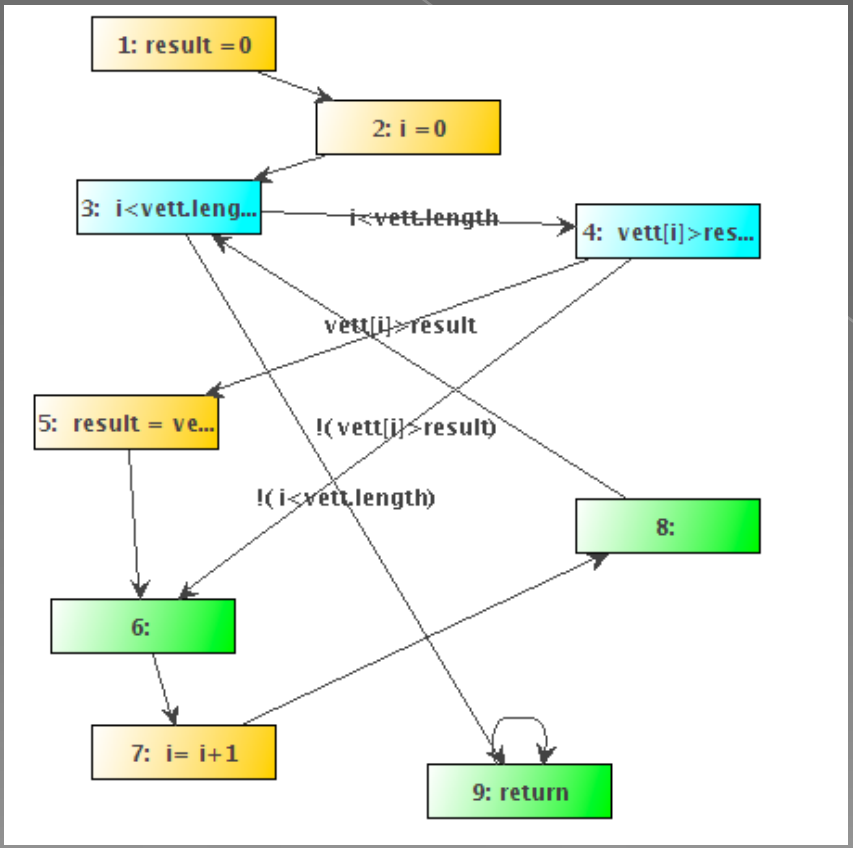
Le 3 tipologie

- Attraversamento semplice
- Cammino ordinato
- Cammino completo

Attraversamento semplice

Dati n nodi $a_1 \ldots a_n$ trova un caso di test che per ogni coppia di nodi consecutivi attraversa almeno una volta l'arco che li congiunge.

Esempio di attraversamento



File: massimo.java

Si sceglie il cammino

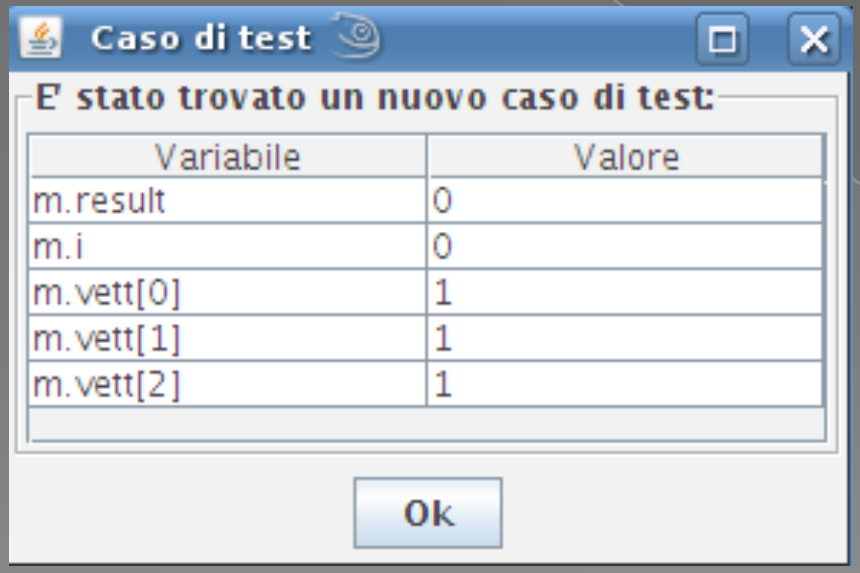
“4 5 6 7 8 3 9”

Esempio di attraversamento 2

LTLSPEC generata:

```
m.PRE -> G(m.PC = 4 -> X m.PC != 5)
   | G(m.PC = 5 -> X m.PC != 6) |
G(m.PC = 6 -> X m.PC != 7) | G(m.PC
 = 7 -> X m.PC != 8) | G(m.PC = 8 ->
   X m.PC != 3) | G(m.PC = 3 -> X
                m.PC != 9)
```

Esempio di attraversamento 3

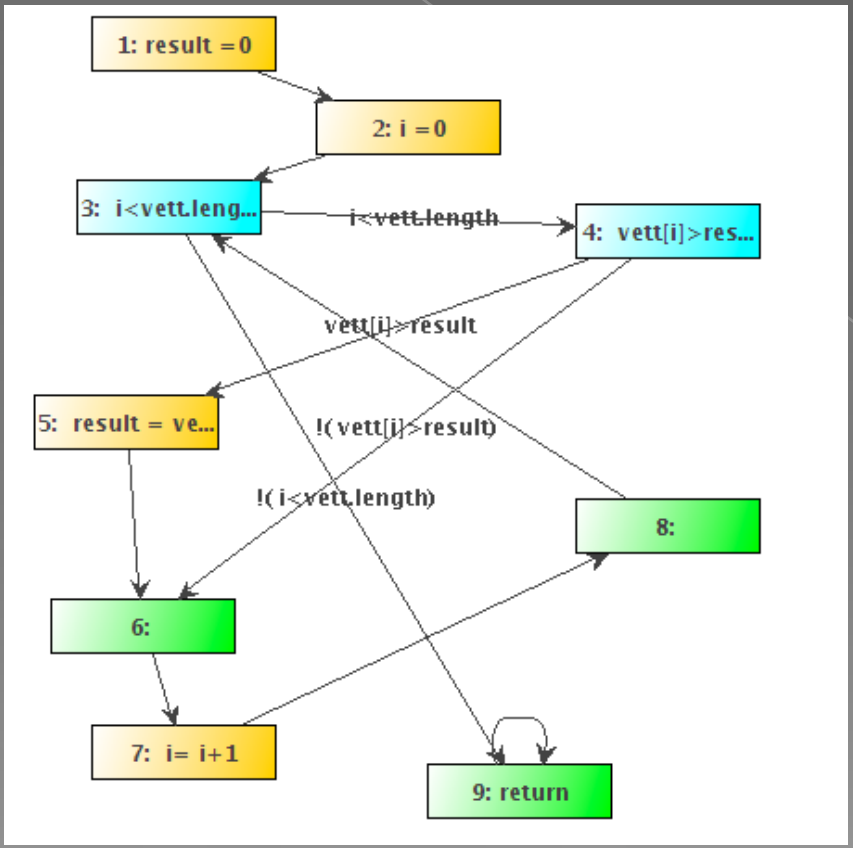


Nel cammino non abbiamo indicato i primi nodi del grafo, ma il caso di test soddisfa comunque la specifica dell'attraversamento

Cammino ordinato

Dati n nodi $a_1 \ldots a_n$ trova un caso di test che passa per ogni nodo e tale che ogni a_i non può essere visitato prima di a_{i-1}.

Esempio cammino ordinato



File: massimo.java

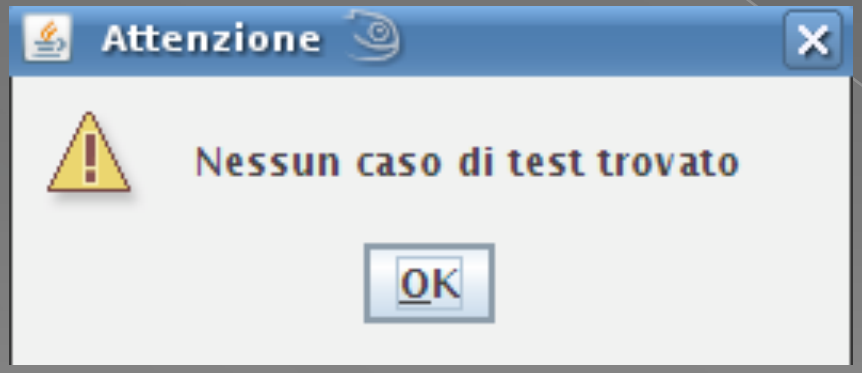
Si sceglie il cammino:
“1 2 4 5 6 7 8 3 9”

Esempio di cammino ordinato 2

LTLSPEC generata:

m.PRE -> (G !(m.PC = 1) | G !(m.PC = 2) | G !(m.PC = 4) | G !(m.PC = 5) | G !(m.PC = 6) | G !(m.PC = 7) | G !(m.PC = 8) | G !(m.PC = 3) | G !(m.PC = 9)) | (G !((m.PC != 2 U m.PC = 1) & (m.PC != 4 U m.PC = 2) & (m.PC != 5 U m.PC = 4) & (m.PC != 6 U m.PC = 5) & (m.PC != 7 U m.PC = 6) & (m.PC != 8 U m.PC = 7) & (m.PC != 3 U m.PC = 8) & (m.PC != 9 U m.PC = 3)))

Esempio cammino ordinato 3



E' violata in ogni caso la condizione per la quale non può essere toccato 3 prima che sia toccato 8

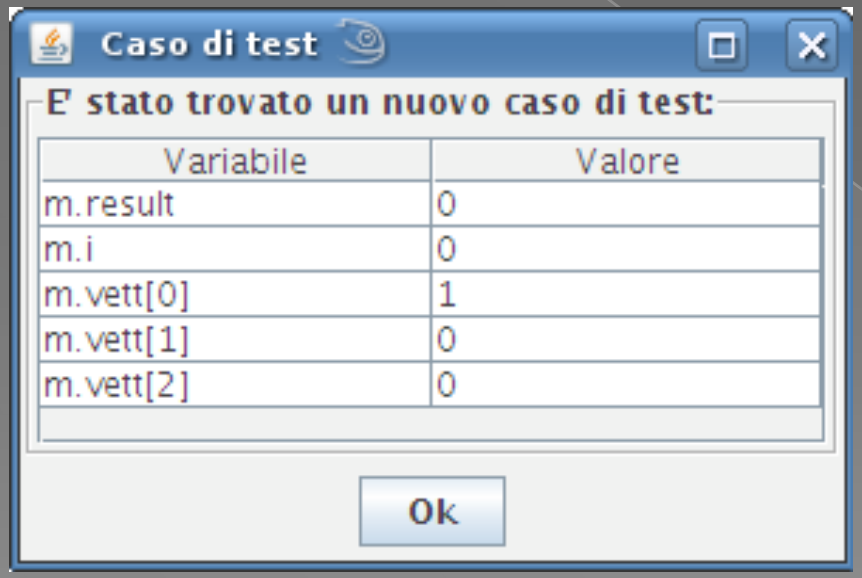
Esempio cammino ordinato 4

Scegliendo invece il cammino “1 2 3 4 5 6 7 8 3 9” si avrà la LTLSPEC:

```
m.PRE -> ( G !(m.PC = 1) | G !(m.PC = 2) | G !(m.PC = 3)
  | G !(m.PC = 4) | G !(m.PC = 5) | G !(m.PC = 6) | G !
 (m.PC = 7) | G !(m.PC = 8) | G !(m.PC = 3) | G !(m.PC =
9) ) | (G !((m.PC != 2 U m.PC = 1) & (m.PC != 3 U m.PC =
 2) & (m.PC != 4 U m.PC = 3) & (m.PC != 5 U m.PC = 4) &
    (m.PC != 6 U m.PC = 5) & (m.PC != 7 U m.PC = 6) &
    (m.PC != 8 U m.PC = 7) & (m.PC != 9 U m.PC = 3)));
```

quindi il nodo 3 può essere toccato prima del nodo 8

Esempio cammino ordinato 4

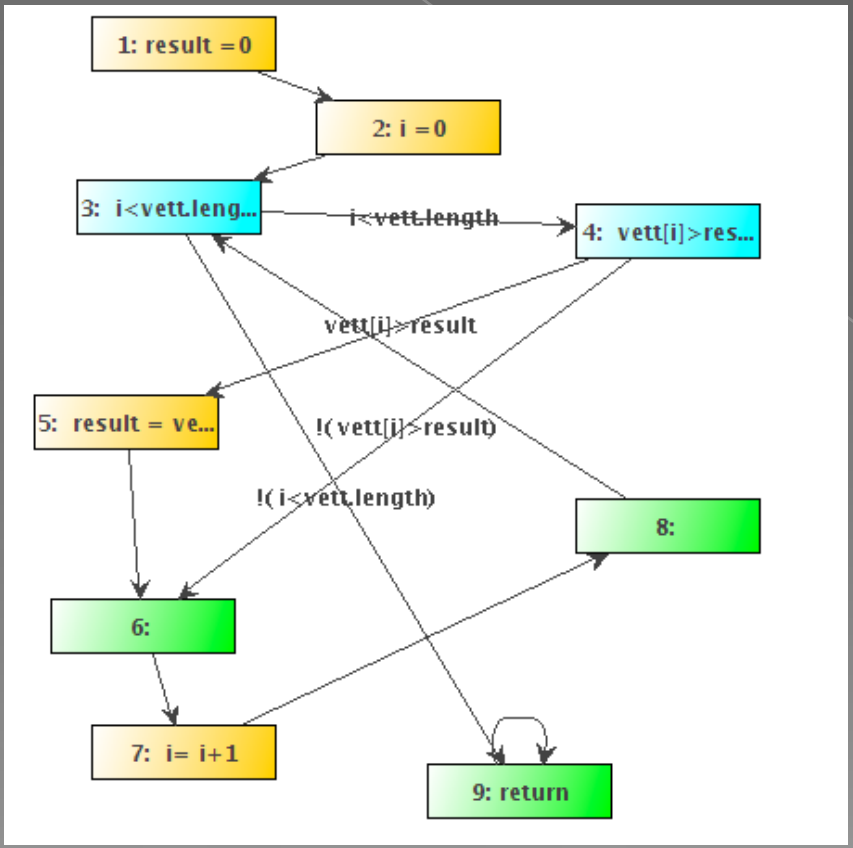


Viene infatti trovato un caso di test

Cammino completo

Dati n nodi $a_1 \ldots a_n$ trova un caso di test che visita gli n nodi nell'esatta sequenza data.

Esempio cammino completo



File: massimo.java

Si sceglie il cammino:

"1 2 3 4 5 6 7 8 3 4 5 6 7 8 3 4 6 7 8 3 9"

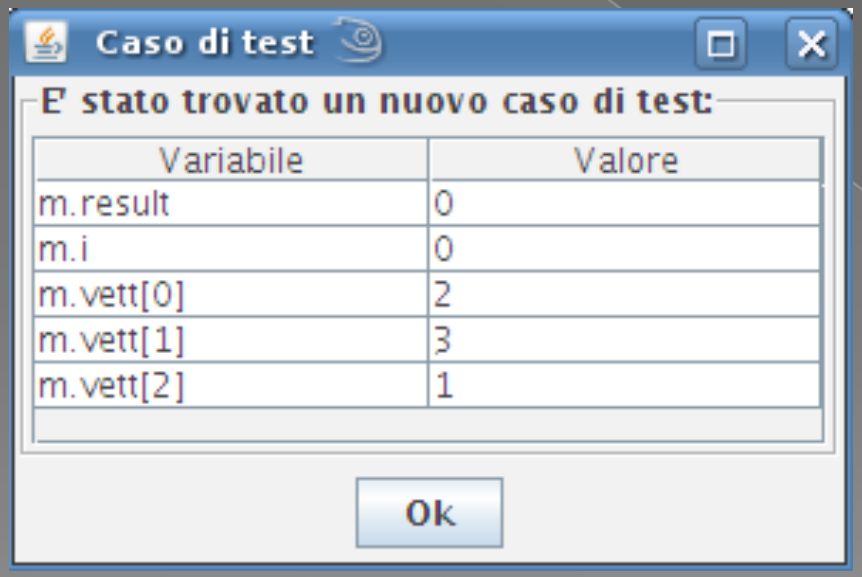
Esempio cammino completo 2

La LTLSPEC sarà:

```
m.PRE -> G !(  m.PC = 1 & X  m.PC = 2 & X X
          m.PC = 3 & X X X  m.PC = 4
                    .......
& X X X X X X X X X X X X X X X X X X X X  m.PC
= 3 & X X X X X X X X X X X X X X X X X X X X X
                    m.PC = 9)
```

Vogliamo quindi che passi 2 volte per il nodo 5 e che quindi result venga aggiornata due volte.

Esempio cammino completo 3



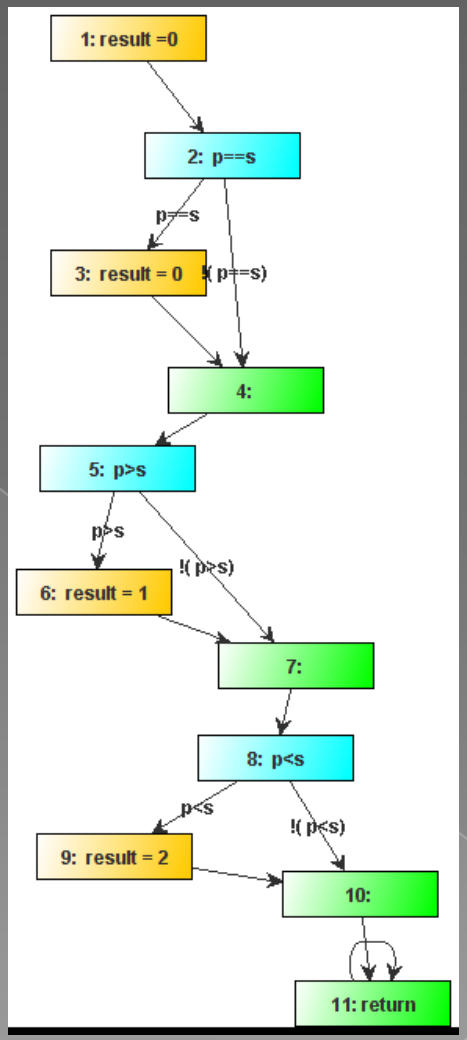
In questo caso di test infatti result viene aggiornata prima con m.vett[0] e poi con m.vett[1]

Esempi

Ecco alcuni esempi di esecuzione di Jasmine su differenti programmi Java in input. Sarà mostrato il grafo di flusso creato e il caso di test trovato a fronte delle diverse richieste dell'utente.

Esempio 1: max(int p, int s)

```
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;
    }

}
```

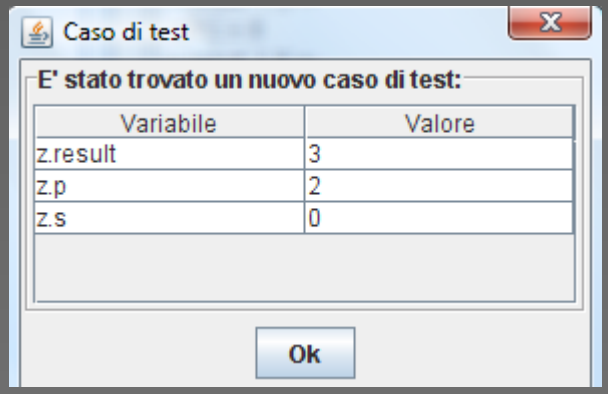


Chiediamo l'attraversamento del cammino 1 2 4 5 6 7 8 10 11

(ovvero p>s)

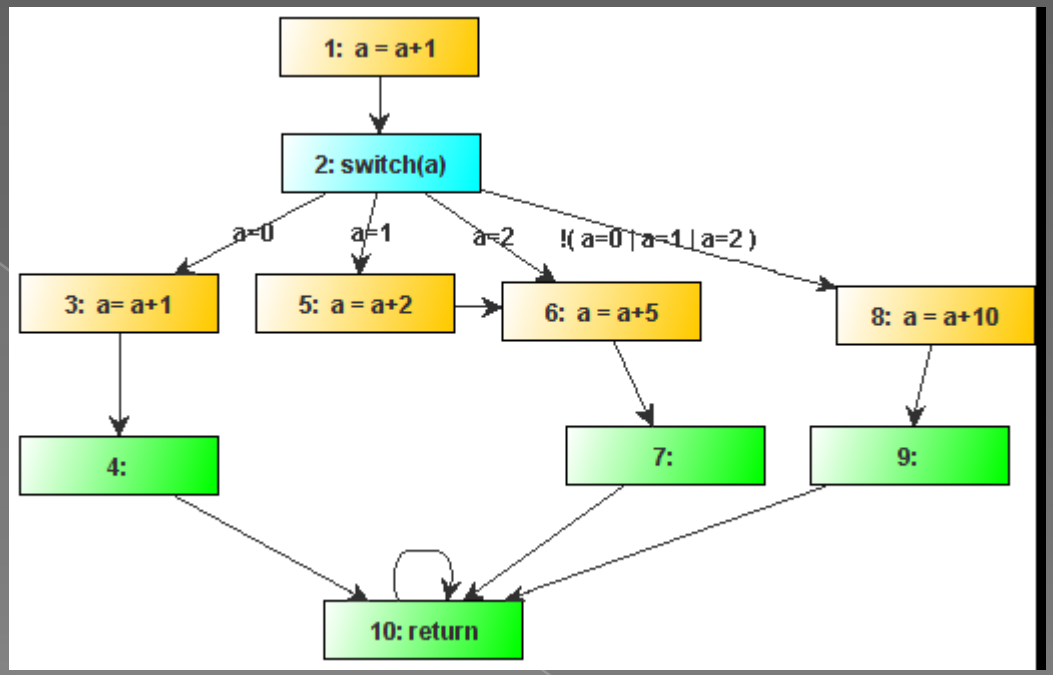
Esempio 1: codice smv

```
MODULE zeroUnoDue_max

VAR
 result : 0..3;
 p : 0..3;
 s : 0..3;
 PC : 1..11;

ASSIGN

DEFINE
 TERM := PC = 11;
 PRE := 1;
 POST := 1;

TRANS
 case
   PC = 1 : next(PC) = 2 & next(result)  =0 & next(p) = p & next(s) = s ;
   PC = 2 &  p=s : next(PC) = 3 & next(result) = result & next(p) = p & next(s) = s ;
   PC = 2 & !( p=s) : next(PC) = 4 & next(result) = result & next(p) = p & next(s) = s ;
   PC = 3 : next(PC) = 4 & next(result)  = 0 & next(p) = p & next(s) = s ;
   PC = 4 : next(PC) = 5 & next(result) = result & next(p) = p & next(s) = s ;
   PC = 5 &  p>s : next(PC) = 6 & next(result) = result & next(p) = p & next(s) = s ;
   PC = 5 & !( p>s) : next(PC) = 7 & next(result) = result & next(p) = p & next(s) = s ;
   PC = 6 : next(PC) = 7 & next(result)  = 1 & next(p) = p & next(s) = s ;
   PC = 7 : next(PC) = 8 & next(result) = result & next(p) = p & next(s) = s ;
   PC = 8 &  p<s : next(PC) = 9 & next(result) = result & next(p) = p & next(s) = s ;
   PC = 8 & !( p<s) : next(PC) = 10 & next(result) = result & next(p) = p & next(s) = s ;
   PC = 9 : next(PC) = 10 & next(result)  = 2 & next(p) = p & next(s) = s ;
   PC = 10 : next(PC) = 11 & next(result) = result & next(p) = p & next(s) = s ;
   PC = 11 : next(PC) = 11 & next(result) = result & next(p) = p & next(s) = s ;
   1: next(PC)=PC & next(p) = p & next(s) = s & next(result) = result ;
esac
--end MODULE zeroUnoDue_max

MODULE main
VAR
 z : zeroUnoDue_max;

ASSIGN
 init(z.PC):=1;
--end MODULE main

LTLSPEC
z.PRE -> G(z.PC = 1 -> X z.PC != 2) | G(z.PC = 2 -> X z.PC != 4) | G(z.PC = 4 -> X z.PC != 5) | G(z.PC = 5 -> X z.PC != 6) | G(z.PC = 6 -> X
        z.PC != 7) | G(z.PC = 7 -> X z.PC != 8) | G(z.PC = 8 -> X z.PC != 10) | G(z.PC = 10 -> X z.PC != 11)
```



Esempio 2: Switch(int a)

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

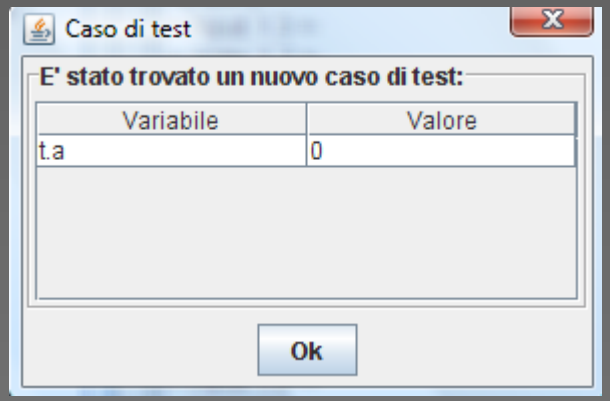


Chiediamo l'attraversamento ordinato del cammino 1 2 5 6 7 10

(notare la prima istruzione: a=a+1, quindi si suppone che a in input debba essere 0)

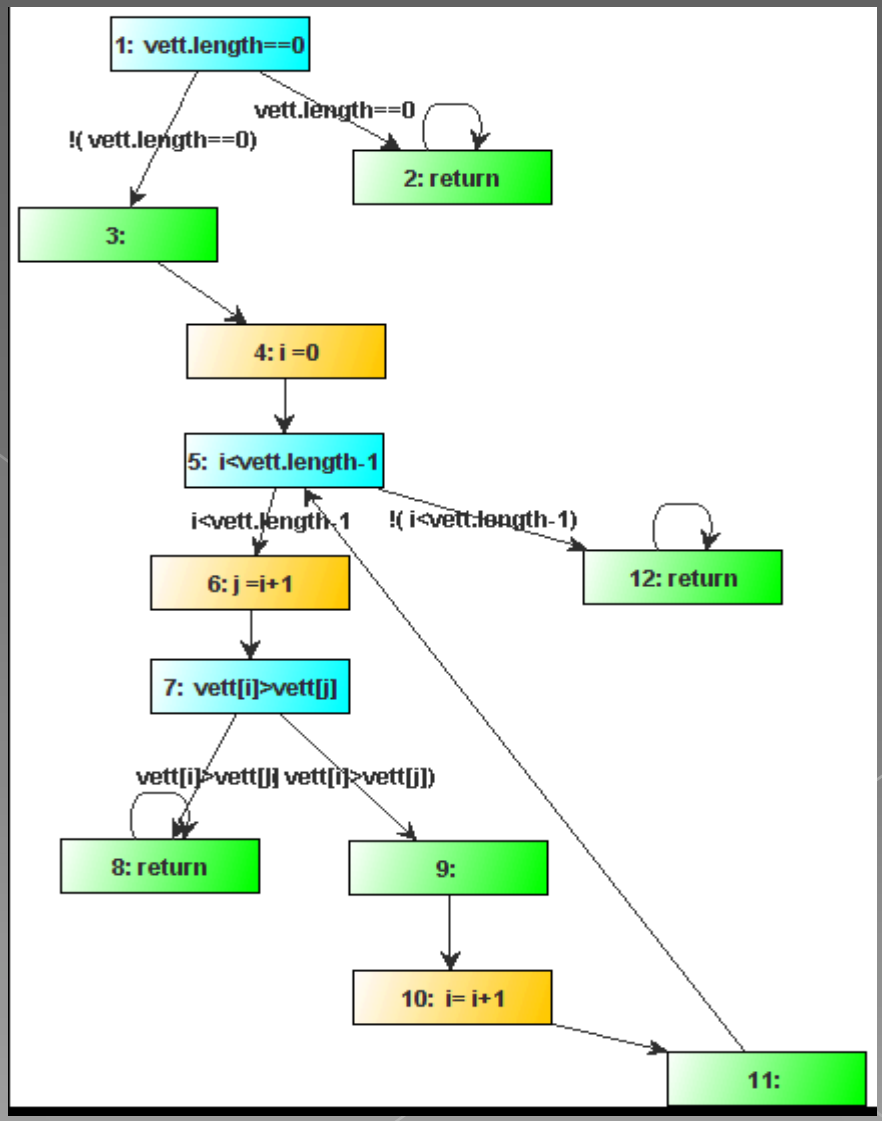
Esempio 2: codice smv

```
MODULE TestSwitch_Switch

VAR
 a : -10..10;
 PC : 1..10;

ASSIGN

DEFINE
 TERM := PC = 10;
 PRE := 1;
 POST := 1;

TRANS
 case
   PC = 1 : next(PC) = 2 & next(a)  = a+1 ;
   PC = 2 &  a=0 : next(PC) = 3 & next(a) = a ;
   PC = 2 &  a=1 : next(PC) = 5 & next(a) = a ;
   PC = 2 &  a=2 : next(PC) = 6 & next(a) = a ;
   PC = 2 & !( a=0 | a=1 | a=2 ) : next(PC) = 8 & next(a) = a ;
   PC = 3 : next(PC) = 4 & next(a) = a+1 ;
   PC = 4 : next(PC) = 10 & next(a) = a ;
   PC = 5 : next(PC) = 6 & next(a)  = a+2 ;
   PC = 6 : next(PC) = 7 & next(a)  = a+5 ;
   PC = 7 : next(PC) = 10 & next(a) = a ;
   PC = 8 : next(PC) = 9 & next(a)  = a+10 ;
   PC = 9 : next(PC) = 10 & next(a) = a ;
   PC = 10 : next(PC) = 10 & next(a) = a ;
   1: next(PC)=PC & next(a) = a ;
esac
--end MODULE TestSwitch_Switch

MODULE main
VAR
 t : TestSwitch_Switch;

ASSIGN
 init(t.PC):=1;
--end MODULE main

LTLSPEC
t.PRE -> ( G !(t.PC = 1) | G !(t.PC = 2) | G !(t.PC = 5) | G !(t.PC = 6) | G !(t.PC = 7) | G !(t.PC = 10) ) | (G !((t.PC != 2 U t.PC = 1) & (t.PC != 5 U
          t.PC = 2) & (t.PC != 6 U t.PC = 5) & (t.PC != 7 U t.PC = 6) & (t.PC != 10 U t.PC = 7)))
```



Esempio 3: isSorted

```
public static boolean isSorted(int vett[]) {
if (vett.length == 0) return true;
for (int i=0; i<vett.length-1; i++) {
        int j=i+1;
        if (vett[i] > vett[j]) return false;
        }
return true;
}
```



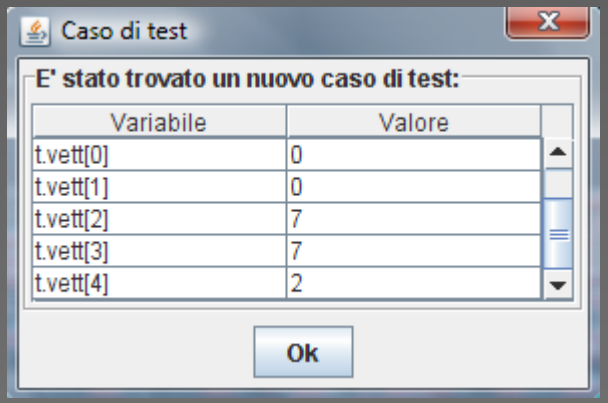
Impostiamo: vett.length=5, range di i, di j e di vett: 0..10

Chiediamo il passaggio per il cammino completo 1 3 4 5 6 7 9 10 11 5 6 7 9 10 11 5 6 7 9 10 11 5 6 7 8

(ovvero un cammino per il quale l'ordinamento non è soddisfatto, ed è il quinto ed ultimo elemento di vett a "rovinare l'ordinamento")

Esempio 3: codice smv

```
MODULE TestDue_isSorted

VAR
 i :0..10;
 j :0..10;
 vett : array 0..4 of 0..10;
 PC : 1..12;

ASSIGN

DEFINE
 TERM := PC = 2 | PC = 8 | PC = 12;
 PRE := 1;
 POST := 1;

TRANS
 case
   PC = 1 &  5=0 : next(PC) = 2 & next(i) = i & next(j) = j & next(vett[0]) = vett[0] & next(vett[1]) = vett[1] & next(vett[2]) = vett[2] & next(vett[3]) = vett[3] & next(vett[4]) = vett[4] ;
   PC = 1 & !( 5=0) : next(PC) = 3 & next(i) = i & next(j) = j & next(vett[0]) = vett[0] & next(vett[1]) = vett[1] & next(vett[2]) = vett[2] & next(vett[3]) = vett[3] & next(vett[4]) = vett[4] ;
   PC = 2 : next(PC) = 2 & next(i) = i & next(j) = j & next(vett[0]) = vett[0] & next(vett[1]) = vett[1] & next(vett[2]) = vett[2] & next(vett[3]) = vett[3] & next(vett[4]) = vett[4] ;
   PC = 3 : next(PC) = 4 & next(i) = i & next(j) = j & next(vett[0]) = vett[0] & next(vett[1]) = vett[1] & next(vett[2]) = vett[2] & next(vett[3]) = vett[3] & next(vett[4]) = vett[4] ;
   PC = 4 : next(PC) = 5 & next(i)  =0 & next(j) = j & next(vett[0]) = vett[0] & next(vett[1]) = vett[1] & next(vett[2]) = vett[2] & next(vett[3]) = vett[3] & next(vett[4]) = vett[4] ;
   PC = 5 &  i<5-1 : next(PC) = 6 & next(i) = i & next(j) = j & next(vett[0]) = vett[0] & next(vett[1]) = vett[1] & next(vett[2]) = vett[2] & next(vett[3]) = vett[3] & next(vett[4]) = vett[4] ;
   PC = 5 & !( i<5-1) : next(PC) = 12 & next(i) = i & next(j) = j & next(vett[0]) = vett[0] & next(vett[1]) = vett[1] & next(vett[2]) = vett[2] & next(vett[3]) = vett[3] & next(vett[4]) = vett[4] ;
   PC = 6 : next(PC) = 7 & next(i) = i & next(j)  =i+1 & next(vett[0]) = vett[0] & next(vett[1]) = vett[1] & next(vett[2]) = vett[2] & next(vett[3]) = vett[3] & next(vett[4]) = vett[4] ;
   PC = 7 &  vett[0]>vett[0] & i=0 & j=0 : next(PC) = 8 & next(i) = i & next(j) = j & next(vett[0]) = vett[0] & next(vett[1]) = vett[1] & next(vett[2]) = vett[2] & next(vett[3]) = vett[3] & next(vett[4])
          = vett[4] ;
   PC = 7 &  vett[1]>vett[0] & i=1 & j=0 : next(PC) = 8 & next(i) = i & next(j) = j & next(vett[0]) = vett[0] & next(vett[1]) = vett[1] & next(vett[2]) = vett[2] & next(vett[3]) = vett[3] & next(vett[4])
          = vett[4] ;
(...)
esac

MODULE main
VAR
 t : TestDue_isSorted;

ASSIGN
 init(t.PC):=1;
--end MODULE main

LTLSPEC
--CAMMINO
t.PRE -> G !(  t.PC = 1 & X  t.PC = 3 & X X  t.PC = 4 & X X X  t.PC = 5 & X X X X  t.PC = 6 & X X X X X  t.PC = 7 & X X X X X X  t.PC = 9 & X X X X X X X  t.PC = 10 & X X X X X X X X  t.PC = 11 &
          X X X X X X X X  t.PC = 5 & X X X X X X X X X  t.PC = 6 & X X X X X X X X X X  t.PC = 7 & X X X X X X X X X X X  t.PC = 9 & X X X X X X X X X X X X  t.PC = 10 & X X X X X
          X X X X X X X X  t.PC = 11 & X X X X X X X X X X X X  t.PC = 5 & X X X X X X X X X X X X X  t.PC = 6 & X X X X X X X X X X X X X X  t.PC = 7 & X X X X X X X X X
          X X X X X X X X X  t.PC = 9 & X X X X X X X X X X X X X X  t.PC = 10 & X X X X X X X X X X X X X X X  t.PC = 11 & X X X X X X X X X X X X X X X X
          t.PC = 5 & X X X X X X X X X X X X X X X X X  t.PC = 6 & X X X X X X X X X X X X X X X X X  t.PC = 7 & X X X X X X X X X X X X X X X X X  t.PC
          = 8)
```



Domande?

Grazie per l'attenzione