

TESTING E VERIFICA DEL SOFTWARE

ESERCITAZIONE 5 – ModelJUnit e Combinatorial Testing

31-5-2017(26-5-2015)

Parte 1: Testing FSM con ModelJUnit

1.1. Predisporre il progetto:

- Scarica modeljunit.jar da: <http://sourceforge.net/projects/modeljunit/> o Oppure lo trovi nello zip della distribuzione
- Crea un progetto (java) eclipse e metti il jar nel build path del progetto (se copi il jar nel progetto puoi farlo col tasto destro)
- Puoi importare le classi di modeljunit senza problemi e scrivere il tuo modello FSM
- Per eseguire il tool, fai doppio click sul jar.
- Il progetto d'esempio si trova sotto "esercizio 0 (On-Off)".

1.2. Scrivere il modello della FSM:

Il modello FSM si scrive direttamente in Java. Normalmente questa classe si collega alla SUT (e non è la SUT stessa) (cioè ha in se un riferimento al sistema under test).

La classe del modello FSM deve estendere l'interfaccia FsmModel con i seguenti metodi (implements FsmModel):

- Object **getState()**: This method returns the current visible state of the EFSM. So this method defines an abstraction function that maps the internal state of the EFSM to the visible states of the EFSM graph. Typically, the result is a string, but it is possible to return any type of object.
- void **reset(boolean)**: This method resets the EFSM to its initial state. When online testing is being used, it should also reset the SUT or create a new instance of the SUT class. The Boolean parameter can be ignored for most unit testing applications.
- @Action void **namei()**: The EFSM must define several of these action methods, each marked with an @Action annotation. These action methods define the transitions of the EFSM. They can change the current state of

the EFSM, and when online testing is being used, they also send test inputs to the SUT and check the correctness of its responses.

- boolean ***name*****iGuard()**: Each action method can optionally have a guard, which is a Boolean method with the same name as the action method but with “Guard” added to the end of the name. When the guard returns true, then the action is enabled (so may be called), and when the guard returns false, the action is disabled (so will not be called). Any action method that does not have a corresponding guard method is considered to have an implicit guard that is always true.

NOTA: il controllo di correttezza del comportamento del SUT viene fatto nei metodi `@Action` e/o `getState`. In `Action` dovrei controllare l'output, in `getState` che il sistema vada nello stato opportuno.

1.3. Ci sono due modi per testare la FSM:

- **ONLINE testing**

(online = la generazione degli input è contemporanea con la loro applicazione alla macchina FSM testata)

Scrivi un caso di test Junit che testa la tua classe. Del tipo:

```
// create our model and a test generation algorithm
Tester tester = new GreedyTester(new FSM());
// build the complete FSM graph for our model, just to ensure
// that we get accurate model coverage metrics.
tester.buildGraph();
// set up our favourite coverage metric
CoverageMetric trCoverage = new TransitionCoverage();
tester.addCoverageMetric(trCoverage);
// ask to print the generated tests
tester.addListener(new VerboseListener());
// generate a small test suite of 20 steps (covers 4/5 transitions)
tester.generate(20);
tester.getModel().println(trCoverage.getName() + " was " +
trCoverage.toString());
```

Puoi cambiare:

1. il tester (random)

2. il misuratore di coverage

Il modo migliore per produrre il codice per l'online testing è usare l'interfaccia del modeljunit.jar.

- **OFFLINE testing** (con l'interfaccia)

Usando il tool come interfaccia grafica possiamo esplorare il modello e generare le tracce offline. Come aprire il progetto con il tool:

1. Scrivi la classe che rappresenta la tua FSM e la tua SUT
2. metti le tue classi in un jar (con export di eclipse)
3. esegui il tool: `java -jar modeljunit-2.5-jar-with-dependencies.jar`

Per far funzionare il tutto devi creare un **nuovo progetto** con jar il tuo jar e classe la tua FSM.

Puoi anche prendere un file di progetto vecchio (.mju) e modificare nel progetto:

```
<className>it.unibg.tvsw.MIACLASSE</className>
```

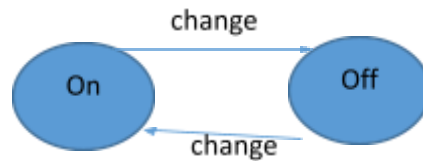
```
<packageLocation>file:esercizio.jar</packageLocation>
```

Dall'interfaccia puoi:

- Esplorare lo spazio degli stati (con generate test)
 - o Cambia tipi di visualizzazione ed esplorazione per ottenere FSM migliori
- Vedere il codice per l'offline testing (utile anche per scoprire errori) con Test configuration
- Animate the model
- Ottenere una misurazione della copertura
- Salvare il progetto (per riaprirlo dopo) .mju

Per fare offline testing si deve esportare i casi di test come testo (come sequenza di metodi action chiamati e stati visitati) e tradurli (manualmente) come casi di test (ad esempio Junit) per il test offline (cioè separato dalla generazione).

Esercizio 0 (On/Off)



Scrivi una classe OnOff che simuli il comportamento di un interruttore, ci sono due stati:

- ON
- OFF

(ricordati di implementare l'interfaccia FsmModel)

Assumi che lo stato iniziale sia OFF.

Prova a scrivere un Online Test.

Esercizio 1 (macchina del caffè):

Un macchina per il caffè riceve in ingresso dei gettoni e produce, quando richiesto il caffè. L'utente può inserire fino a 5 gettoni. Esistono due tipi di caffè:

- Normale (1 gettone);
- Super (2 gettoni).

Il credito attuale viene visualizzato sul display della macchina.

Formalizza (su carta):

- ☐ gli stati della macchina;
- ☐ l'input che accetta la macchina (es: token, produci caffè e reset);
- ☐ le transizioni di stato;

Prova a implementare la macchinetta in Java, con gli opportuni metodi e output per sapere quanti crediti ci sono e così via.

Implementa la sua specifica FSM in Java.

Prova ad eseguire l'online testing con il tool.

Prova anche a generare tracce con il tool.

Esercizio 2 (tema d'esame 27 Gennaio 2015):

Un magazzino contiene cinque tipi di prodotti. Il magazzino può contenere al massimo 100 unità di ogni prodotto. Si possono aggiungere quantità di un certo prodotto ma al massimo 10 alla volta e solo fino a raggiungere il livello massimo.

Considera il caso semplificato di un magazzino con solo 2 prodotti in quantità da 0 a 5 a cui si possono aggiungere solo 1 o 2 prodotti.

Modella l'evoluzione del sistema con una FSM. Implementa la FSM con modelJunit.

Parte 2: partition and combinatorial testing

2.1 Partition testing:

Esercizio 3 (equazioni di secondo grado):

Dato un metodo che restituisce il numero di soluzioni per una equazione di secondo grado:

```
static public int numSolutions(int a, int b, int c){....}
```

- implementa il metodo
- Fai input domain modeling: modella l'insieme degli input in modo da coprirli secondo diversi criteri di copertura. Partiziona l'insieme `int x int x int` (triple) in modi diversi:
 1. INTERFACE-BASED: considera una partizione per `a`, per `b` e per `c` e prendi dei valori significativi (però senza considerare il problema) per ognuna delle partizioni. Scrivi i casi di test in JUnit per tutte le combinazioni delle partizioni.
 2. FUNCTIONALITY-BASED: dividi il dominio in partizioni a seconda del risultato che ti aspetti dal metodo che devi testare. Prendi un rappresentante per partizione e scrivi i casi di test in JUnit.

2.2 Combinatorial testing con citlab:

Scarica citlab: <http://svn.code.sf.net/p/citlab/code/updatesite/>

Puoi trovare esempi di sintassi

<https://code.google.com/a/eclipselabs.org/p/citlab/wiki/CitLabLanguage>

Esercizio 4:

Applica il combinatorial testing al precedente modello (interface based approach) usando CITLAB. (Prova a generare il pairwise usando citlab + ACTS)

Esercizio fatto in classe - RushHour

Si vuole scrivere un modello e una implementazione semplificata per il puzzle RushHour.

Nel RushHour una griglia 6x6 contiene 6 macchine numerate da 1 a 6 (ogni macchina per semplicità ha dimensione 1x1). Puoi pensare di rappresentare la griglia come array bidimensionale di interi con le celle vuote indicate con 0 oppure rappresentare la posizione delle 6 macchine con una coppia di interi da 0 a 5 per ogni macchina.

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | | | | 6 | | |
| 2 | | | | | | 2 |
| 3 | | | 1 | | | 3 |
| 4 | | | | | | 4 |
| 5 | | 5 | | | | |
| 6 | | | | | | |

Obiettivo del gioco è portare la macchina numero 1 (chiamata macchina rossa) all'uscita della griglia, che corrisponde alla cella con indici (3,6) (o 2,5 se le coordinate iniziano da 0) indicata dallo sfondo scuro. Quando la macchina rossa raggiunge l'uscita il gioco termina.

La configurazione iniziale del gioco è quella indicata.

Il sistema deve permettere lo spostamento di una macchina in una casella adiacente libera. Deve inoltre essere possibile sapere se la macchina rossa è all'uscita.

Java

La classe deve avere come minimo un metodo boolean `moveCar` con segnatura:

```
moveCar(int row, int col, int dir)
```

che permetta di muovere una macchina sulla griglia; il metodo ritorna `true` se la macchina è stata spostata, `false` altrimenti. I parametri `row` e `col` indicano una cella della griglia contenente la macchina che si vuole spostare; `dir` indica una direzione nel seguente modo:

1. verso l'alto;
2. verso destra;
3. verso il basso;
4. verso sinistra.

Il metodo deve controllare che i valori passati siano corretti. Se una macchina non si può spostare in direzione `dir` perché ostruita da un'altra macchina o dai limiti della griglia, non deve essere spostata.

C'è anche il metodo **redCarAtExit** che dice se la macchina rossa ha raggiunto l'uscita;

In Java ti consiglio di rappresentare la griglia come un array bidimensionale.

INPUT DOMAIN MODELLING

Consegna: Applica IDM al metodo di spostamento delle macchine. Individua gli input da passare e spiega le scelte che hai fatto. Scrivi i corrispondenti casi di test Junit.

Soluzione 1: Interface based: considera ogni parametro a se e poi applica la copertura di tutte le combinazioni possibili

Soluzione 2: functionality-based: trova le caratteristiche da testare studiando il problema

Soluzione 1: Per IDM ho fatto un caso di test guardando anche un po' le funzionalità del metodo.

Prendo i parametri secondo questa tabella che individua delle partizioni

| | | | |
|-----|----|-----------|-----|
| row | <0 | Tra 0 e 5 | >=6 |
| col | <0 | Tra 0 e 5 | >=6 |
| dir | <1 | Tra 1 e 4 | >4 |

Prendo per tutti un valore non valido < dei valori validi, un valore valido, e un valore non valido > dei valori validi. In totale 9 combinazioni che sono le 27 chiamate dei test che ho scritto in Junit.

Per le partizioni prendo questi valori (che sono sul limite delle partizioni)

| | | | |
|-----|----|---|---|
| row | -1 | 1 | 6 |
| col | -1 | 1 | 6 |
| dir | 0 | 1 | 5 |

Soluzione 2:

Considero questa partizione degli input:

| | Partizione | |
|----|--|--|
| B1 | Sposto una macchina in una cella sbagliata (non esiste) | |
| B2 | Sposto una macchina da una cella in cui non c'è macchina | |
| B3 | Sposto una macchina all'interno del puzzle (ma non all'uscita) | |
| B4 | Cerco di spostare una macchina fuori dal puzzle | |
| B5 | Sposto la macchina rossa all'uscita | |

COMBINATORIAL TESTING

Consegna: Applica il combinatorial testing al sistema semplificato con solo 3 auto in modo di avere tutte le configurazioni possibili. Prova a generare la copertura pairwise delle posizioni. Includi i necessari constraints. Se riesci aggiungi anche una variabile che indica se l'auto rossa è nella posizione di uscita.

Scrivo i parametri per le tre macchine tenendo separate righe e colonne. Aggiungo un parametro che dice se la prima macchina è in uscita. Aggiungo i constraints per evitare che le macchine si sovrappongano.

Model RushHour

Parameters:

// prima macchina righe e colonne

Range r1 [1..6];

Range c1 [1..6];

// seconda macchina righe e colonne

Range r2 [1..6];

Range c2 [1..6];

// terza macchina righe e colonne

Range r3 [1..6];

Range c3 [1..6];

// prima macchine in uscita

Boolean redAtExit;

end

Constraints:

// la prima macchina è in uscita se e solo se è in 1,6

redAtExit <=> (r1==3 and c1==6)

// car1 e car2 non possono occupare la stessa cella

r1!=r2 or c1!=c2

// idem per le altre

r1!=r3 or c1!=c3

r2!=r3 or c2!=c3

end

Genero il pairwise e ottengo 54 casi di test:

| Test | r1 | c1 | r2 | c2 | r3 | c3 | redAtExit |
|------|----|----|----|----|----|----|-----------|
| 36 | 6 | 6 | 1 | 1 | 2 | 3 | false |
| 37 | 5 | 1 | 2 | 1 | 1 | 2 | false |
| 38 | 1 | 2 | 3 | 2 | 2 | 1 | false |
| 39 | 4 | 4 | 4 | 3 | 3 | 1 | false |
| 40 | 4 | 2 | 6 | 5 | 2 | 2 | false |
| 41 | 2 | 3 | 1 | 6 | 1 | 1 | false |
| 42 | 3 | 1 | 6 | 4 | 1 | 3 | false |
| 43 | 5 | 6 | 3 | 6 | 4 | 6 | false |
| 44 | 6 | 1 | 2 | 4 | 1 | 5 | false |
| 45 | 1 | 4 | 4 | 1 | 6 | 2 | false |
| 46 | 6 | 5 | 6 | 6 | 4 | 1 | false |
| 47 | 4 | 5 | 6 | 5 | 4 | 3 | false |
| 48 | 3 | 6 | 5 | 1 | 1 | 2 | true |
| 49 | 6 | 4 | 1 | 3 | 1 | 4 | false |
| 50 | 3 | 6 | 5 | 2 | 2 | 6 | true |
| 51 | 3 | 6 | 1 | 3 | 3 | 3 | true |
| 52 | 3 | 6 | 2 | 4 | 4 | 4 | true |
| 53 | 3 | 6 | 6 | 6 | 6 | 5 | true |
| 54 | 3 | 6 | 3 | 4 | 6 | 2 | true |

Esercizio magazzino

Un magazzino contiene **cinque** tipi di prodotti. Il magazzino può contenere al massimo **100** unità di ogni prodotto. Si possono aggiungere quantità di un certo prodotto ma al massimo **10** alla volta e solo fino a raggiungere il livello massimo.

Java

In Java scrivi una classe Magazzino che implementa il sistema sopra. Ha tre metodi:

`boolean insert(int productIndex, int addQuantity)`

dati un indice di prodotto `productIndex` e una quantità `addQuantity`, aggiunge il valore `addQuantity` (al massimo 10 prodotti); al prodotto identificato da `productIndex` Il metodo ritorna `true` se l'aggiunta viene eseguita, `false` altrimenti. L'aggiunta non viene eseguita se l'indice di prodotto è errato, o se la quantità aggiunta non è corretta (non compresa tra 1 e 10), o se la nuova quantità che si otterrebbe con l'aggiunta supera le 100 unità.

`boolean isFull(int productIndex)`

dice se un certo prodotto ha raggiunto il massimo

`boolean isFull()`

restituisce se il magazzino è completamente pieno (tutti i prodotti sono la massimo)

Le quantità di prodotti a magazzino sono memorizzate in un array di interi, dove ogni cella corrisponde ad un prodotto. All'inizio il magazzino è completamente vuoto.

IDM

Prova a modellare l'input al metodo `boolean insert(int productIndex, int addQuantity)`

Prova sia l'approccio

- interface-based (ignora il significato di `productIndex` e `addQuantity`)
- functionality based in cui individui i criteri con cui scegliere i valori dei due parametri

COMB (Combinatorial testing)

Rappresenta in combinatorial testing la chiamata del metodo

`boolean insert(int productIndex, int addQuantity)`

Considera come variabili del problema:

`productIndex` e `addQuantity` che sono i parametri del metodo

`returnValue` che il valore ritornato

`nproductsOld` che rappresenta il numero di prodotti prima della chiamata

`nproductsNew` che rappresenta il numero di prodotti dopo la chiamata

Introduci i domini opportuni e i vincoli tra i parametri. In questo modo hai anche l'oracolo.

Applica il combinatorial testing. Prova a generare la copertura pairwise. Prova a tradurre una riga dal tuo test in un test Junit per la classe Magazzino.