

Documentazione

Sommario

Esercizio 1 - TESTING	4
Scenario AllarmeTest	4
Copertura istruzioni (Statement coverage)	5
Copertura archi/decisioni (Branch coverage/Decision coverage)	6
Copertura MCDC (Modified Condition/decision coverage)	7
Copertura totale	8
Randoop	9
Esercizio 2 - JML	10
Precondizioni (obblighi per il cliente)	10
Postcondizioni (obblighi per il fornitore)	10
spec_public	10
Invarianti (condizione che è sempre vera)	10
Violazione di precondizioni	11
Violazione di postcondizioni	11
Violazione di invarianti	12
Esercizio 3 - KEY	13
Esercizio 4 - NUSMV	14

Esame di Testing e verifica del software del 6/6/2016 di Ghisleni Stefano 1020581

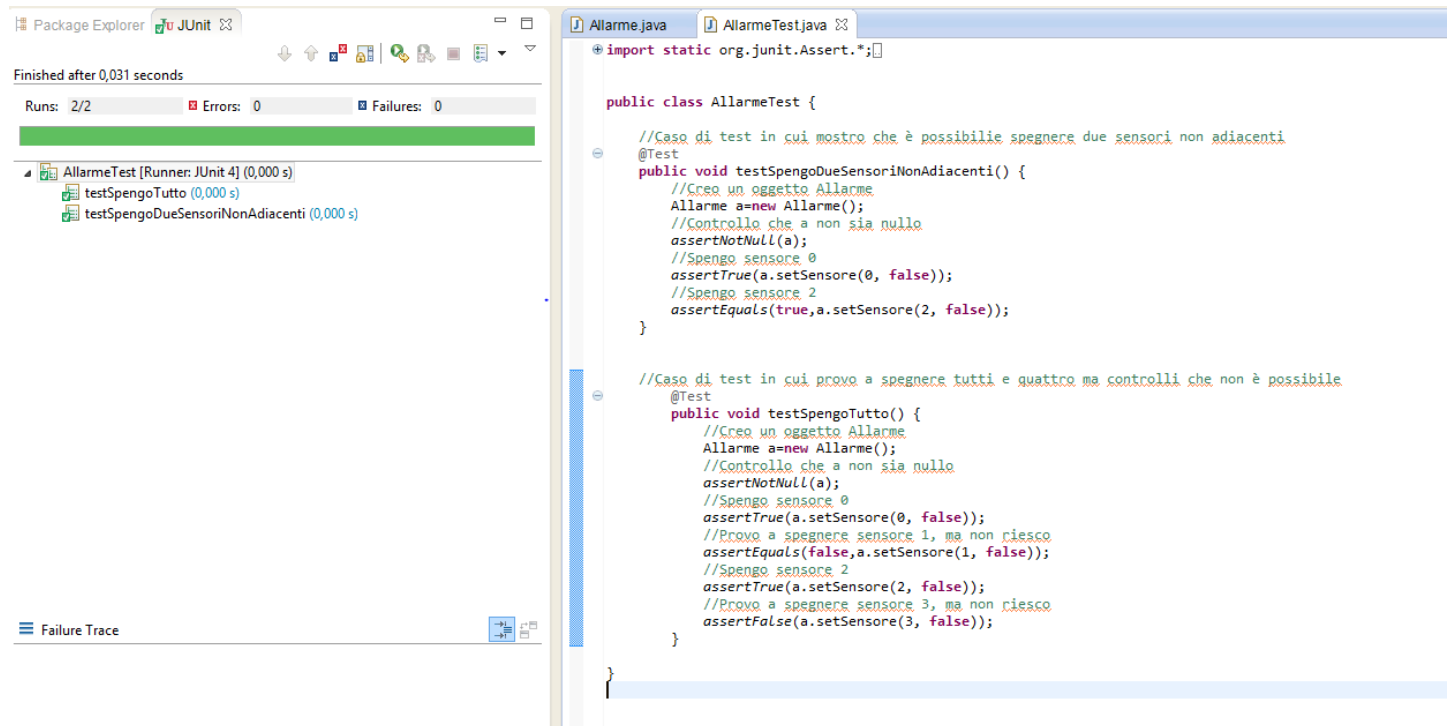
Esercizio 5 - FSM.....	17
ONLINE testing.....	17
OFFLINE testing (con l'interfaccia).....	18
Prendo queste transizioni e provo a tradurle in un caso di test Junit.....	19
Esercizio 6 - COMBINATORIAL TESTING - CITLAB	20
Copertura Pair-Wise:	20

Esercizio 1 - TESTING

L'esercizio 1 consisteva nella realizzazioni della classe ... in grado di simulare funzionamento del ... descritto nel tema d'esame e testarla attraverso il tool JUnit.

Scenario AllarmeTest

Scrivo un caso di test relativo allo scenario AllarmeTest, lo eseguo con JUnit e controllo il risultato.



Il risultato, come possiamo notare dalla barra verde, è positivo.

Copertura istruzioni (Statement coverage)

Scrivo un caso di test relativo alla copertura delle istruzioni e lo eseguo con CodeCover.

The screenshot displays an IDE with the following components:

- Package Explorer:** Shows the project structure with 'AllarmeTestIstruzioni' and 'Allarme.java'.
- JUnit Runner:** Indicates the test is finished after 0.703 seconds with 1/1 runs, 0 errors, and 0 failures.
- Code Editor:** Contains the Java code for 'Allarme.java'. The code includes a constructor, a 'setSensore' method, an 'accesso' method, and a 'pericolo' method. The 'setSensore' method is highlighted in green, indicating it was executed.
- Coverage Tab:** Shows the test results for 'Statement Coverage'. The table below summarizes the data:

Name	Statement	Branch	Loop	Term	?-Operator	Synchronized
GHISLENI_TESTING	100,0 %	100,0 %	?	66,7 %	?	?
Allarme	100,0 %	100,0 %	?	66,7 %	?	?
Allarme	100,0 %	-	?	100,0 %	?	?
accesso	100,0 %	-	?	-	?	?
pericolo	100,0 %	-	?	-	?	?
setSensore	100,0 %	100,0 %	?	60,0 %	?	?

La copertura relativa alle istruzioni (chiamata Statement in CodeCover) è del 100%.

Copertura archi/decisioni (Branch coverage/Decision coverage)

Scrivo un caso di test relativo alla copertura delle decisioni/archi e lo eseguo con CodeCover.

The screenshot shows an IDE with a JUnit test class `AllarmeTestDecisioniBranch.java` and the CodeCover coverage report.

JUnit Test Code:

```
import static org.junit.Assert.*;

//Copertura decisioni/branch
//N.B la copertura delle decisioni è sempre equivalente alla copertura dei branch
//Di fatto riutilizzo lo stesso caso di test della copertura delle istruzioni, visto che già quella aveva
//copertura dei branch del 100%
public class AllarmeTestDecisioniBranch {

    @Test
    public void test() {
        // Crea un oggetto Allarme
        Allarme a = new Allarme();
        // Controllo che a non sia nullo
        assertNotNull(a);
        //Spegno sensore 0
        assertTrue(a.setSensore(0, false));
        //Provo a spegnere sensore -1 (che però non esiste)
        assertFalse(a.setSensore(-1, false));
        //Verifico che il sensore 1 non sia acceso
        assertFalse(a.acceso(0));
        //Verifico che non vi è pericolo
        assertFalse(a.pericolo());
    }
}
```

CodeCover Coverage Report:

Name	Statement	Branch	Loop	Term	?-Operator	Synchronized
GHISLENI_TESTING	100,0 %	100,0 %	?	66,7 %	?	?
Allarme	100,0 %	100,0 %	?	66,7 %	?	?
Allarme	100,0 %	-	?	100,0 %	?	?
acceso	100,0 %	-	?	-	?	?
pericolo	100,0 %	-	?	-	?	?
setSensore	100,0 %	100,0 %	?	60,0 %	?	?

La copertura relativa alle decisioni/archi (chiamata Branch in CodeCover) è del 100%.

Copertura MCDC (Modified Condition/decision coverage)

Nel MCDC il test set deve essere preso in modo che ogni condizione all'interno di una decisione deve far variare in modo indipendente il valore finale della decisione.

Scrivo un caso di test relativo alla copertura MCDC della decisione

```
(sensore<0 || sensore>=N || sensori[(sensore+N-1)%N]==false || sensori[(sensore+1)%N]==false ||
accesso==sensori[sensore])
```

e lo eseguo con CodeCover.

Osservando il Boolean Analyzer vediamo che vengono valutati:

- un caso in cui avendo tutte le condizioni false la decisione finale fa sì che si abbia `return true`;
- gli altri casi in cui una singola condizione è vera e tutte le altre condizioni sono false (o non valutate a causa della `short circuit evaluation`), in questi casi si ottiene che la decisione finale fa sì che si abbia `return false`.

```
import static org.junit.Assert.*;

//Copertura MDCD
public class AllarmeTestMCDC {

    @Test
    public void test() {
        // Creo un oggetto Allarme
        Allarme a = new Allarme();
        //Considero la seguente decisione
        //(sensor<=0 || sensore>N) OR sensori[sensore+1]N!=false || accesso==sensori[sensore])

        //Caso in cui tutte le condizioni sono a false
        assertTrue(a.setSensore(0, false));

        //Caso in cui tutte le condizioni sono a false tranne sensore<0 che è a true
        assertFalse(a.setSensore(-1, false));

        //Caso in cui tutte le condizioni sono a false tranne sensore==4 che è a true
        assertFalse(a.setSensore(4, false));

        //Caso in cui tutte le condizioni sono a false tranne sensori[(sensore+1)N]!=false che è a true
        assertFalse(a.setSensore(1, false));

        //Caso in cui tutte le condizioni sono a false tranne sensori[(sensore+1)N]=false che è a true
        assertTrue(a.setSensore(3, false));

        //Caso in cui tutte le condizioni sono a false tranne accesso==sensori[sensore] che è a true
        assertTrue(a.setAccesso(true));
    }
}
```

Quindi effettivamente abbiamo copertura al 100% di MCDC della decisione

$(\text{sensore} < 0 \parallel \text{sensore} \geq N \parallel \text{sensori}[(\text{sensore} + N - 1) \% N] == \text{false} \parallel \text{sensori}[(\text{sensore} + 1) \% N] == \text{false} \parallel \text{acceso} == \text{sensori}[\text{sensore}])$. Inoltre si ha il minor numero di casi di test per la copertura MCDC pari a $n+1$, dove n è il numero di condizioni all'interno della decisione. In questo caso abbiamo 5 condizioni e 6 casi di test.

Copertura totale

Copro tutto: Statement, Branch e Term.

The screenshot shows an IDE with the following components:

- Package Explorer:** Shows the project structure with 'JUnit' and 'AllarmeTestCoperturaTotale'.
- JUnit Runner:** Shows 'Finished after 0,719 seconds', 'Runs: 2/2', 'Errors: 0', and 'Failures: 0'.
- Code Editor:** Displays the Java code for 'AllarmeTestCoperturaTotale'.
- Test Results:** Shows a table of test results for 'AllarmeTestCoperturaTotale'.

Code Editor Content:

```
import static org.junit.Assert.*;

// Copro tutto: Statement, Branch e Term
public class AllarmeTestCoperturaTotale {

    @Test
    public void test1() {
        // Creo un oggetto Allarme
        Allarme a = new Allarme();
        // Controllo che a non sia nullo
        assertNotNull(a);
        // Spengo sensore 0
        assertTrue(a.setSensore(0, false));
        // Provo a spegnere sensore -1 (che però non esiste)
        assertFalse(a.setSensore(-1, false));
        // Verifico che il sensore 1 non sia acceso
        assertFalse(a.acceso());
        // Verifico che non vi è pericolo
        assertFalse(a.pericolo());
    }

    @Test
    public void test2() {
        // Creo un oggetto Allarme
        Allarme a = new Allarme();
        // Considero la seguente decisione
        // (sensore < 0 || sensore >= N || sensori[(sensore + N - 1) % N] == false ||
        // sensori[(sensore + 1) % N] == false || acceso == sensori[sensore])

        // Caso in cui tutte le condizioni sono a false
        assertTrue(a.setSensore(0, false));

        // Caso in cui tutte le condizioni sono a false tranne sensore < 0 che è a
        // true
        assertFalse(a.setSensore(-1, false));
    }
}
```

Test Results Table:

Name	Statement	Branch	Loop	Term	?-Operator	Synchronized
GHISLENI_TESTING	100,0 %	100,0 %	?	100,0 %	?	?
Allarme	100,0 %	100,0 %	?	100,0 %	?	?
Allarme	100,0 %	-	?	100,0 %	?	?
acceso	100,0 %	-	?	-	?	?
pericolo	100,0 %	-	?	-	?	?
setSensore	100,0 %	100,0 %	?	100,0 %	?	?

Randoop

Genero in modo automatico con il tool randoop 49 casi di test. Controllo la copertura con CodeCover sui casi di test generati da randoop e osservo che la copertura risulta essere 100% degli Statement, Branch e Term.

The screenshot shows an IDE with the Randoop tool interface on the left and the CodeCoverage coverage tool on the right.

Randoop Interface (Left):

- Package Explorer: JUnit, Randoop
- Finished after 0,75 seconds
- Runs: 49/49, Errors: 0, Failures: 0
- Test Results: RandoopTest0 [Runner: JUnit 4] (0,672 s)

CodeCoverage Interface (Right):

- Problems, Javadoc, Declaration, Console, Test Sessions, Coverage, Boolean Analyzer
- Show methods with: Statement Coverage, 90,5 %

CodeCoverage Coverage Table:

Name	Statement	Branch	Loop	Term	?-Operator	Synchronized
GHISLENI_TESTING	100,0 %	100,0 %	?	100,0 %	?	?
Allarme	100,0 %	100,0 %	?	100,0 %	?	?
Allarme	100,0 %	-	?	100,0 %	?	?
accesso	100,0 %	-	?	-	?	?
pericolo	100,0 %	-	?	-	?	?
setSensore	100,0 %	100,0 %	?	100,0 %	?	?

Source Code (AllarmeTestCoperturaTotale.java):

```

assertFalse(a.setSensore(-1, false));
// Verifico che il sensore 1 non sia acceso
assertFalse(a.accesso(0));
// Verifico che non vi è pericolo
assertFalse(a.pericolo());
}

@Test
public void test2() {
    // Crea un oggetto Allarme
    Allarme a = new Allarme();
    // Considero la seguente decisione
    // (sensore<0 || sensore>=N || sensori[(sensore+N-1)%N]==false ||
    // sensori[(sensore+1)%N]==false || acceso==sensori[sensore])

    // Caso in cui tutte le condizioni sono a false
    assertTrue(a.setSensore(0, false));

    // Caso in cui tutte le condizioni sono a false tranne sensore<0 che è a
    // true
    assertFalse(a.setSensore(-1, false));

    // Caso in cui tutte le condizioni sono a false tranne sensore>=4 che è
    // a true
    assertFalse(a.setSensore(4, false));

    // Caso in cui tutte le condizioni sono a false tranne
    // sensori[(sensore+N-1)%N]==false che è a true
    assertFalse(a.setSensore(1, false));

    // Caso in cui tutte le condizioni sono a false tranne
    // sensori[(sensore+1)%N]==false che è a true
    assertFalse(a.setSensore(3, false));

    // Caso in cui tutte le condizioni sono a false tranne

```

Esercizio 2 - JML

In questo esercizio introduciamo l'uso di contratti all'interno della nostra classe Java, per fare ciò usiamo JML.

Precondizioni (obblighi per il cliente)

Converto tutte le condizioni date dagli if all'interno dei metodi in precondizioni. Per comodità lascio anche gli if di controllo dentro i metodi (anche se non più necessarie, visto che adesso sono sostituite dalle precondizioni) per poter eseguire le violazioni di precondizioni senza generare eccezioni Java.

Guardare il codice per vedere le precondizioni aggiunte.

Postcondizioni (obblighi per il fornitore)

Guardare il codice per vedere le postcondizioni aggiunte.

spec_public

Ai campi privati della classe aggiungo `/*@ spec_public @*/`.

Invarianti (condizione che è sempre vera)

Guardare il codice per vedere gli invarianti aggiunti alla classe.

Dopo aver scritto i contratti scrivo una classe Main nella quale testare il corretto funzionamento della classe e provo a violare precondizioni, postcondizioni, invarianti.

Violazione di precondizioni

Per poter violare una precondizione chiamiamo dal main un metodo della classe con contratti passando come parametri dei valori che non rispettano la precondizione.

```
D:\TESTING\GHISLENI_STEFANO\GHISLENI_JML\src\Main.java:13: JML precondition is false
    a.setSensore(-1, false);
    ^
D:\TESTING\GHISLENI_STEFANO\GHISLENI_JML\src\Allarme.java:37: Associated declaration: D:\TESTING\GHISLENI_STEFANO\GHISLENI_JML\src\Main.java:13:
    //@ requires sensore>=0 && sensore<N && sensori[(sensore+N-1)%N]==true && sensori[(sensore+1)%N]==true && acceso!=sensori[sensore];
    ^

D:\TESTING\GHISLENI_STEFANO\GHISLENI_JML\src\Main.java:14: JML precondition is false
    a.accesso(-1); //Violo requires sensore>=0 && sensore<N;
    ^
D:\TESTING\GHISLENI_STEFANO\GHISLENI_JML\src\Allarme.java:60: Associated declaration: D:\TESTING\GHISLENI_STEFANO\GHISLENI_JML\src\Main.java:14:
    //@ requires sensore>=0 && sensore<N;
    ^
```

Violazione di postcondizioni

Per poter violare una postcondizione introduciamo un errore nel codice della classe, in modo tale che la postcondizione non sia più vera.

```
D:\TESTING\GHISLENI_STEFANO\GHISLENI_JML\src\Allarme.java:22: JML postcondition is false
    public Allarme()
    ^
D:\TESTING\GHISLENI_STEFANO\GHISLENI_JML\src\Allarme.java:21: Associated declaration: D:\TESTING\GHISLENI_STEFANO\GHISLENI_JML\src\Allarme.java:22:
    //@ensures (\forallall int i ; i >= 0 && i < N ; sensori[i] == true);
    ^

D:\TESTING\GHISLENI_STEFANO\GHISLENI_JML\src\Allarme.java:89: JML postcondition is false
    public boolean acceso(int sensore)
    ^
D:\TESTING\GHISLENI_STEFANO\GHISLENI_JML\src\Allarme.java:88: Associated declaration: D:\TESTING\GHISLENI_STEFANO\GHISLENI_JML\src\Allarme.java:89:
    //@ensures \result ==> (sensore>=0 && sensore<N && sensori[sensore]);
    ^

D:\TESTING\GHISLENI_STEFANO\GHISLENI_JML\src\Allarme.java:99: JML postcondition is false
    public boolean pericolo()
    ^
D:\TESTING\GHISLENI_STEFANO\GHISLENI_JML\src\Allarme.java:98: Associated declaration: D:\TESTING\GHISLENI_STEFANO\GHISLENI_JML\src\Allarme.java:99:
    //@ensures \result ==> (\forallall int i; i >= 0 && i < N; !sensori[i] && !sensori[(i+1)%N]);
    ^
```

Violazione di invarianti

Per poter violare un invariante introduciamo un errore nel codice della classe, in modo tale che l'invariante non sia più vero.

```
D:\TESTING\GHISLENI_STEFANO\GHISLENI_JML\src\Allarme.java:11: Associated declaration: D:\TESTING\GHISLENI_STEFANO\GHISLENI_JML\src\Main.java:7:
//@ public invariant sensori != null;
      ^
```

```
..
D:\TESTING\GHISLENI_STEFANO\GHISLENI_JML\src\Allarme.java:17: Associated declaration: D:\TESTING\GHISLENI_STEFANO\GHISLENI_JML\src\Allarme.java:58:
//@ public invariant (\exists int i; 0 <= i && i < 4; sensori[i]);
      ^
```

Esercizio 3 - KEY

Dopo aver scritto i contratti necessari al corretto funzionamento del programma nell'esercizio precedente li verifichiamo in maniera formale, ossia dimostriamo la correttezza parziale dal punto di vista logico/matematico dei contratti in maniera automatica con il tool MonKey (non dimostriamo che i metodi terminano).

Aggiungo `//@ diverges true`; ai contratti per non costringerci a provare che i metodi terminano.

Per dimostrare i loop è stata usata la configurazione di Key con Loop treatment → Expand.

N.B. Per chiudere `public boolean pericolo()` ho modificato

```
//@ensures \result ==> (\forall int i; i >= 0 && i < N; !sensori[i] && !sensori[(i+1)%N]);
```

in

```
//@ensures \result ==> (!sensori[0] && !sensori[1]) || (!sensori[1] && !sensori[2]) || (!sensori[2] && !sensori[3]) ||
(!sensori[3] && !sensori[0]);
```

Probabilmente Key fa fatica a gestire %

The screenshot shows the MonKey Eclipse plugin interface. The 'Proofs' tab is active, displaying a table of proof results for the 'Allarme' method. The table has columns for Type, Target, Contract, Proof Reuse, Proof Result, Nodes, Branches, Time (milliseconds), and G... (Goal). The 'Proof Result' column shows 'Closed' for all four entries. The 'Nodes' column shows values 616, 3786, 377, and 1981. The 'Branches' column shows values 19, 120, 12, and 31. The 'Time (milliseconds)' column shows values 2000, 9453, 1378, and 4063. The 'G...' column shows 'G...' for all four entries. The status bar at the bottom indicates 'Sym | Proof Result = 4 / 4 Closed, Nodes = 6760, Branches = 182, Time (milliseconds) = 17094, Time (milliseconds) = Load & Time (milliseconds) = 20328'.

Type	Target	Contract	Proof Reuse	Proof Result	Nodes	Branches	Time (milli...	G...	G...
Allarme	Allarme()	JML operation contract 0	New Proof	Closed	616	19	2000		
Allarme	pericolo()	JML operation contract 0	New Proof	Closed	3786	120	9453		
Allarme	accesso(int)	JML operation contract 0	New Proof	Closed	377	12	1378		
Allarme	setSensore(int, boolean)	JML operation contract 0	New Proof	Closed	1981	31	4063		

Sym | Proof Result = 4 / 4 Closed, Nodes = 6760, Branches = 182, Time (milliseconds) = 17094, Time (milliseconds) = Load & Time (milliseconds) = 20328

Open Key | Load selected Proofs | Start selected Proofs | Save selected Proofs | Export Proofs | Help

Il risultato ottenuto è molto buono dato che ogni metodo viene chiuso (dimostrato) correttamente dai contratti che ho scritto.

Esercizio 4 - NUSMV

In questo esercizio utilizziamo il tool NuSMV per descrivere il modello che rappresenta il funzionamento di Sensori Allarme.

Ho definito alcune specifiche LTL/CTL per poter eseguire la verifica formale di alcune proprietà che il modello deve avere (Model Checking).

P1

```
-- specification AG !pericolo  is true
```

P2

```
-- specification !(EF (sensore0.statoSensore = FALSE & sensore2.statoSensore = FALSE))  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  sensore0.statoSensore = TRUE
  sensore0.accendoSpengo = FALSE
  sensore1.statoSensore = TRUE
  sensore1.accendoSpengo = FALSE
  sensore2.statoSensore = TRUE
  sensore2.accendoSpengo = FALSE
  sensore3.statoSensore = TRUE
  sensore3.accendoSpengo = FALSE
  sceltaSensore = 0
  pericolo = FALSE
-> State: 1.2 <-
  sensore0.statoSensore = FALSE
  sceltaSensore = 2
-> State: 1.3 <-
  sensore2.statoSensore = FALSE
  sceltaSensore = 0
```

P3

```

-- specification AG (statoSensore = FALSE -> AF statoSensore) IN sensore3 is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 4.1 <-
  sensore0.statoSensore = TRUE
  sensore0.accendoSpengo = FALSE
  sensore1.statoSensore = TRUE
  sensore1.accendoSpengo = FALSE
  sensore2.statoSensore = TRUE
  sensore2.accendoSpengo = FALSE
  sensore3.statoSensore = TRUE
  sensore3.accendoSpengo = FALSE
  sceltaSensore = 0
  pericolo = FALSE
-> State: 4.2 <-
  sensore0.statoSensore = FALSE
  sensore0.accendoSpengo = TRUE
-> State: 4.3 <-
  sensore0.statoSensore = TRUE
  sensore0.accendoSpengo = FALSE
  sceltaSensore = 3
-- Loop starts here
-> State: 4.4 <-
  sensore3.statoSensore = FALSE
  sceltaSensore = 0
-> State: 4.5 <-

```

P4

```

-- specification AG (statoSensore = TRUE -> AF statoSensore = FALSE) IN sensore3 is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-- Loop starts here
-> State: 4.1 <-
  sensore0.statoSensore = TRUE
  sensore0.accendoSpengo = FALSE
  sensore1.statoSensore = TRUE
  sensore1.accendoSpengo = FALSE
  sensore2.statoSensore = TRUE
  sensore2.accendoSpengo = FALSE
  sensore3.statoSensore = TRUE
  sensore3.accendoSpengo = FALSE
  sceltaSensore = 0
  pericolo = FALSE
-> State: 4.2 <-
  sensore0.statoSensore = FALSE
  sensore0.accendoSpengo = TRUE
-> State: 4.3 <-
  sensore0.statoSensore = TRUE
  sensore0.accendoSpengo = FALSE

```

P5

```
-- specification AG ((sensore0.statoSensore = TRUE & sensore1.statoSensore = FALSE) -> A [ sensore0.statoSensore = TRUE U sensore1.statoSensore = TRUE ] ) is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  sensore0.statoSensore = TRUE
  sensore0.accendoSpengo = FALSE
  sensore1.statoSensore = TRUE
  sensore1.accendoSpengo = FALSE
  sensore2.statoSensore = TRUE
  sensore2.accendoSpengo = FALSE
  sensore3.statoSensore = TRUE
  sensore3.accendoSpengo = FALSE
  sceltaSensore = 0
  pericolo = FALSE
-> State: 1.2 <-
  sensore0.statoSensore = FALSE
  sensore0.accendoSpengo = TRUE
-> State: 1.3 <-
  sensore0.statoSensore = TRUE
  sensore0.accendoSpengo = FALSE
  sceltaSensore = 1
-- Loop starts here
-> State: 1.4 <-
  sensore1.statoSensore = FALSE
  sensore1.accendoSpengo = TRUE
  sceltaSensore = 0
-> State: 1.5 <-
```

Oltre alle specifiche richieste vengono inserite delle proprietà di safety e liveness.

```
-- Altre proprietà

-- Safety:
-- è sempre vero che sensore0 è acceso oppure sensore1 è acceso
CTLSPEC AG (sensore0.statoSensore | sensore1.statoSensore); -- proprietà vera

-- Liveness
-- esiste uno stato futuro in cui sensore0 è spento
CTLSPEC EF sensore0.statoSensore=FALSE; -- proprietà vera

-- specification AG (sensore0.statoSensore | sensore1.statoSensore) is true
-- specification EF sensore0.statoSensore = FALSE is true
```


Esercizio 5 - FSM

Modello l'evoluzione del sistema con una FSM e la implemento con ModelJUnit.

ONLINE testing

Nell'Online testing la generazione degli input è contemporanea con la loro applicazione alla macchina FSM testata.

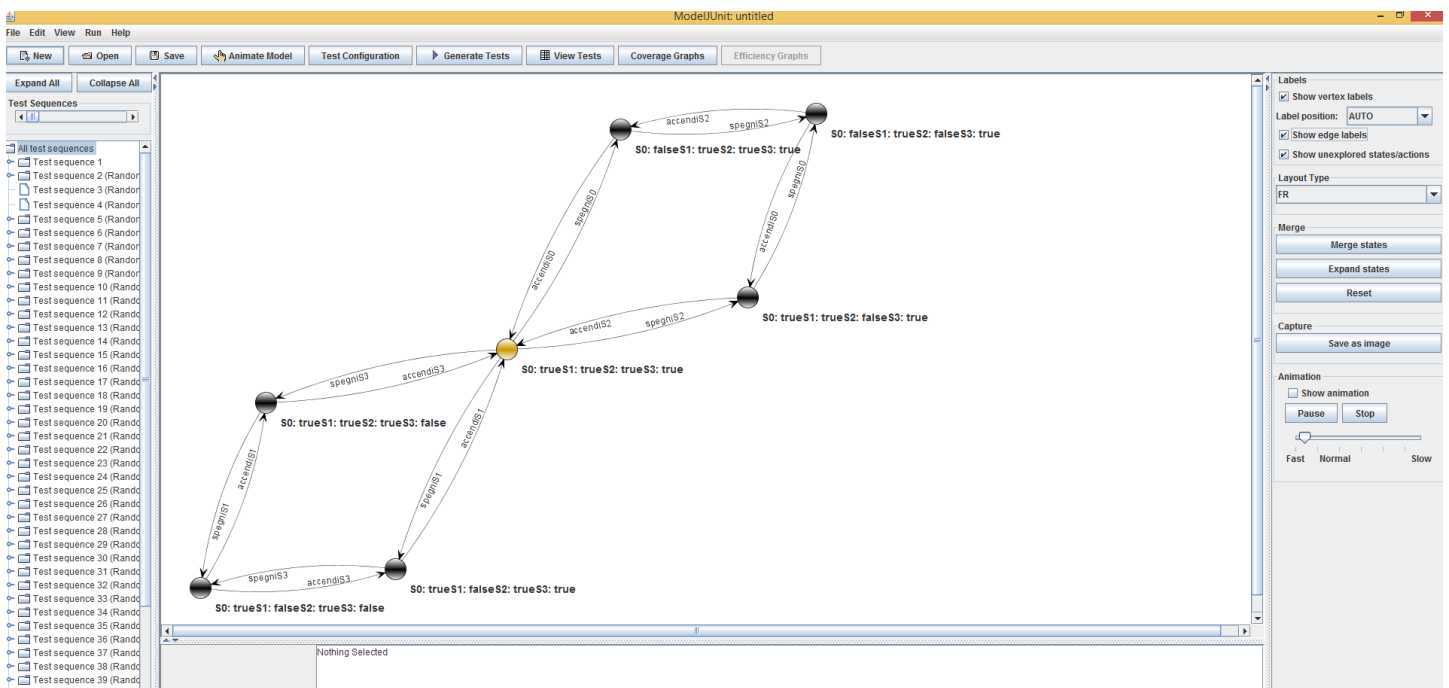
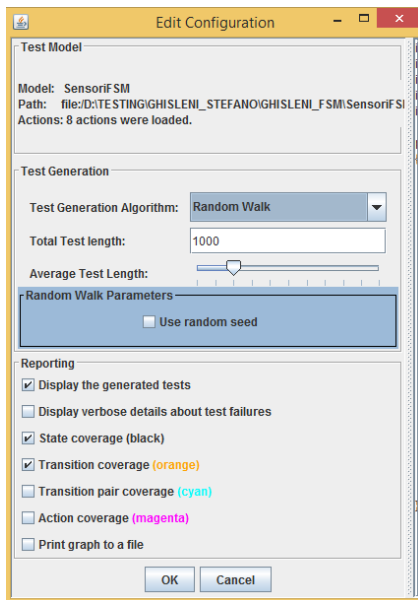
The screenshot shows the Eclipse IDE with the following components:

- Editor:** Displays the `TestOnline.java` file. The code imports classes from `nz.ac.waikato.modeljunit` and `nz.ac.waikato.modeljunit.coverage`. It defines a `TestOnline` class with a `main` method that creates a `SensorIFSM` model, Greedy and Random testers, and coverage metrics. Comments in Italian describe the online testing process.
- Task List:** Shows a task named "TestOnline" with a sub-task "main(String[]): void".
- Console:** Displays the output of the `TestOnline` application. It shows a series of test cases (e.g., "done (S0: falseS1: trueS2: falseS3: true, accendiS2, S0: falseS1: trueS2: trueS3: true)") and coverage results:
 - Copertura degli stati con GreedyTester = 7/7
 - Copertura delle transizioni con GreedyTester = 16/16
 - Copertura degli stati con RandomTester = 7/7
 - Copertura delle transizioni con RandomTester = 16/16

Come possiamo notare otteniamo una copertura totale sia degli stati che delle transizioni, sia utilizzando un metodo di generazione di casi di test Greedy sia utilizzando metodo di generazione di casi di test Random.

OFFLINE testing (con l'interfaccia)

Usando il tool come interfaccia grafica possiamo esplorare il modello e generare le tracce offline.



Prendo queste transizioni e provo a tradurle in un caso di test Junit

done (S0: trueS1: trueS2: trueS3: true, spegniS3, S0: trueS1: trueS2: trueS3: false)

done (S0: trueS1: trueS2: trueS3: false, accendiS3, S0: trueS1: trueS2: trueS3: true)

done (S0: trueS1: trueS2: trueS3: true, spegniS1, S0: trueS1: falseS2: trueS3: true)

done (S0: trueS1: falseS2: trueS3: true, accendiS1, S0: trueS1: trueS2: trueS3: true)

done (S0: trueS1: trueS2: trueS3: true, spegniS0, S0: falseS1: trueS2: trueS3: true)

Il test è superato, quindi abbiamo un po' di fiducia che il modello sia conforme alla classe Java scritta nell'esercizio Testing.

Esercizio 6 - COMBINATORIAL TESTING - CITLAB

Utilizzo il tool CitLab per effettuare il Combinatorial testing. Una volta definiti parametri e vincoli del modello, utilizzo l'estensione ACTS IpoF per generare i valori dei parametri. In particolar modo ci sono diversi criteri di scelta delle combinazioni di valori. Con il criterio Pair-Wise Coverage si ha che ogni valore di ogni parametro deve essere combinato con ogni valore di ogni altro parametro (combinio i valori a coppie di parametri).

Copertura Pair-Wise:

The screenshot shows the CitLab IDE interface. The top pane displays the `Sensori.citl` file with the following content:

```

1 Model Sensori
2
3 //Parametri
4 Parameters:
5   //Sensori
6   Boolean s0;
7   Boolean s1;
8   Boolean s2;
9   Boolean s3;
10 //Situazione di pericolo
11 Boolean pericolo;
12 end
13
14 //Vincoli
15 Constraints:
16 //Se un sensore è spento allora i sensori adiacenti sono accesi
17 # s0==false => (s3==true and s1==true) #
18 # s1==false => (s0==true and s2==true) #
19 # s2==false => (s1==true and s3==true) #
20 # s3==false => (s2==true and s0==true) #
21 //Vi è pericolo se e solo se vi sono due sensori adiacenti spenti
22 # pericolo==true <=> ((s0==false and s1==false) or (s1==false and s2==false) or (s2==false and s3==false) or (s3==false and s0==false)) #
23 end
24

```

The bottom pane shows the test results table with 6 tests generated. The table has columns for Test, s0, s1, s2, s3, and pericolo.

Test	s0	s1	s2	s3	pericolo
1	true	true	false	true	false
2	true	false	true	false	false
3	false	true	true	true	false
4	false	true	false	true	false
5	true	false	true	true	false
6	true	true	true	false	false

On the right side of the bottom pane, there is a configuration panel for the test session:

- Name: IpoF
- Time: 0.641
- 6
- Export button
- Summary: Sensori IpoF 2016/06/06 12:31:29