Assignment 4
Model-Based Testing from FSM Models

Model-Based Testing
DIT848/GU and TDA260/Chalmers

April, 2013

1 Introduction

The goals of this assignment are for you to learn how to design simple finite state machine (FSM) models, and to understand and apply several algorithms for generating tests from FSM models. In this assignment you will generate test sequences from your FSM model by hand, but will write a simple test harness that can execute the tests automatically.

This assignment covers the first four stages of the model-based testing lifecycle: modelling, test generation, test concretization and test execution (see p27 of the textbook). Record the amount of time you spend on each of these stages so that you can analyze this in your conclusions. You should read sections 3.3.1 and 5.1 of the textbook as background for this assignment.

2 Submitting your work

If you want to have feedback on your assignment, check with Hamid Ebadi (`hamide #@# chalmers.se`). If you want to submit, please attach a .zip or .tar.gz archive, containing your source code and .txt, .pdf or .doc file describing your answers. For subject of your email and also to name your archive file, please use the assignment number and your (last) name as in the following example: `assignment04_names.zip`.
The deadline for this assignment is **Wednesday, 1 May 2013**.

3 Modelling

Design one or more FSM models of the Java `Set interface` (see `http://java.sun.com/j2se/1.5.0/docs/api/java/util/Set.html`), assuming that your system under test (SUT) will be some implementation of `Set<String>`. Each state of your FSM will correspond to a particular state of a Set implementation. For example, you might start with just four FSM states as follows:

FSM states

FSM state	sut.size()	Strings in the sut set
Empty	0	
abc	1	"abc"
abc+xyz	2	"abc", "xyz"
xyz	1	"xyz"

Each transition of your FSM should model an action that can be implemented by calling one of the methods in the Set class, or a sequence of those methods. For example, the transition that goes from the "Empty" state to the "abc" state corresponds to calling the `SUT.add("abc")` method. This transition might be written as "addabc/true", to show that the string 'abc" is added and the expected result from the method call is true. Some other transitions in your FSM might correspond to calling a long sequence of Set update methods.

You can design your FSM on paper, or using a drawing program such as the drawing tools in Microsoft Word.

You may want to design more than one model, in order to test different aspects of the behaviour of the `Set` interface.

4 Test Generation

Generate tests from those models by manually applying a variety of graph traversal algorithms, including at least the following:

- An all-states traversal.
- An all-transitions traversal. That is, the Chinese Postman Algorithm.

You could write each test sequence as a sequence of space-separated triples:

```
transitionName   expectedOutput   newState
transitionName   expectedOutput   newState
transitionName   expectedOutput   newState
...              ...             ...
```

5 Test Harness

To execute the test sequences that you have generated, you should write a small Java program that reads a sequence from a file or from standard input and executes the corresponding calls on a `Set` implementation. (The directory **myset** contains 3 buggy implementations of `Set<String>`: `MySet1.java, MySet3.java, MySet3.java`.)

One way of doing this is to use the Java 5.0 `Scanner` class to read the three strings from each line, then branch to a separate block of code for each transition name. (You can use a series of if-else to do this, or a switch statement if you define your transition names as an enum and convert the input strings into enum values – see `http://www.xefer.com/2006/12/switchonstring`). For example, the block of code for the "addabc" transition might look like this (`sut` is the `Set<String>` object that is being tested).

```
case ADDABC:
    boolean result = sut.add("abc");
    if (expectedOutput.equals("true"))
       checkEqual(true, result);
    else
       checkEqual(false, result);
    checkInState(newState);
    break;
```

Note that the `expectedOutput` and `newState` strings come from the triple on the current input line. The `checkInState` method branches to some code for each state, which checks that the contents or size of the set agrees with the expected FSM state.

6 Test Execution and Conclusions

Execute each of your tests on at least two different implementations of `SetSet<String>` (`HashSet`, `CopyOnWriteArraySet`, `LinkedHashSet`, `TreeSet`, etc.) and on the buggy `Set` implementations that you have received. Present the summary results of your test executions in a table. You do not need to write an incident report for each failure.

Discuss your conclusions about the effectiveness and the cost of using this style of model-based testing (with FSM models and manual test generation). Discuss the relative amount of time you spent on each phase and how that might be different if: (A) you had some tools that could do the test generation automatically, and (B) you had to do regression testing on ten successive releases of a new `Set` implementation.

7 Acknowledgment, Copyright and Disclaimer

This assignment is based on material provided by *Prof. Mark Utting* (`http://www.cs.waikato.ac.nz/~marku`).