Assignment 4
Model-based testings

Model-Based Testing
DIT848/GU and TDA260/Chalmers

Spring 2014

Introduction

The purpose of this assignment is to exercise writing your own Extended Finite-State Machine (EFSM) models for software testing.

Submitting your work

You have to submit your work through the Fire system `http://xdat09.ce.chalmers.se/mbt/`. Upload your source code and txt or pdf file describing your answers.
The deadline for this assignment is **Wednesday 18 May 2014**.

1 Graph Theory Techniques

A Finite-State Machine (FSM) can be represented as a *state transition table*: a matrix specifying for each input symbol what are the states connected by the corresponding symbol. There are different ways of representing such tables, for instance by writing all the states in the first column and the events in the first row, as shown below. This table is the state transition table for an FSM designed to test the calculator from assignment 1. E.g. the row containing $Initial$, $Read_1$, $-$, $-$, $Initial$ represents the following transitions: $Initial \xrightarrow{digit} Read_1$ (i.e. from state $Initial$ to state $Read_1$ with input $digit$). If the result state is "$-$" in the table, it means that there is no transition from a state with the corresponding given input in the table (e.g., there is no transition from $Read_1$ with input "=").

	digit	*operator*	`=`	`C`
$Initial$	$Read_1$	$-$	$-$	$Initial$
$Read_1$	$Read_1$	Op	$-$	$Initial$
Op	$Read_2$	Op	$Result$	$Initial$
$Read_2$	$Read_2$	Op	$Result$	$Initial$
$Result$	$Read_1$	Op	$-$	$Initial$

Given the state table above, answer the following questions ($Initial$ is the initial state).

1.1 Part 1

Is the given FSM:

- deterministic?
- initially connected?
- complete?
- strongly connected?

Note: Read page 148 (chapter 5, section 5.1.2) of Uttang & Legeard book *Practical Model-Based Testing* if you are not aware of the above terminology about graphs.

1.2 Part 2

Let us consider the following input sequences:

Input 1: *digit operator digit* `=` `C`
Input 2: `C` *digit digit* `C` *digit operator operator digit operator* `C` *digit operator digit digit* `C` *digit operator digit* `=` *digit operator digit* `=` *operator* `=` `C`

Indicate which of the following coverage criteria on the model are satisfied by the above input sequences:

- All-states coverage
- All-configurations coverage
- All-transitions coverage
- All-transition-pairs coverage
- All-loop-free-paths coverage
- All-round-trips coverage
- All-paths coverage
- All input values

2 Test Generation

Generate tests from the model given previously 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 pairs:

transitionInput	newState
transitionInput	newState
transitionInput	newState
...	...

2.1 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 the original `Calculator` implementation. One way of doing this is to use the Java 5.0 `Scanner` class to read the two 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 "digit" transition might look like this (`sut` is the `Calculator` object that is being tested):

```
%\begin{minted}{java}
    case DIGIT:
        boolean result = sut.pressDigit("1");
        checkInState(newState);
        break;
%\end{minted}
```

Note that the `newState` strings come from the pairs 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.

2.2 Test Execution and Conclusions

Execute each of your tests on the `Calculator` implementation from Assignment 01. 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 `Calculator` implementation.

3 Model-based testing with ModelJUnit

In this part, you are given an implementation of a calculator (`Calculator2.java`) based on the specification in 3.1. Design one or more EFSM models for this calculator assuming that it is your system under test (SUT). You can use the FSM given in the first part of this assignment as a starting point.

3.1 Specification

This calculator is an extension of the one that is used in the first assignment where it is also possible to have parenthesized expressions. The informal specification for this calculator is explained in what follows.
The calculator has the following buttons:

- digits: `0` `1` `2` `3` `4` `5` `6` `7` `8` `9`
- operations: `+` `-` `*` `/`
- execute: `=`
- reset: `C`
- parantheses: `(` `)`
- memory buttons: `M` `MR` `MC`

The calculator behaves mostly like the one from previous assignments except for the following points.

- If an open parenthesis button is pressed a new subexpression is started, where both operands and operators are entered normally. When a close parenthesis is pressed. this subexpression is evaluated and then the result is stored as an operand.
- Open parenthesis is not allowed after a digit, and close parenthesis is not allowed after an operator. An operator is not allowed after an open parenthesis, and a digit is not allowed after a close parenthesis.
- If the open parenthesis button is immediately followed by a close parenthesis, it should be an error.
- It is not possible to press `=` within a subexpression.
- If an operation button is pressed while there is an operand and operation memorized, the calculator acts as if `=` was pressed before the operation button.
- If an operation button is pressed exactly after another operation button, the last one will be used for calculation and the first operation button will be ignored.
- In case of a division by 0, the buffer should show `Err`.

This calculator has an additional state holding element: the memory.

- When the calculator is first turned on, the memory is empty.
- At any time, pressing the `MC` empties the memory.
- The memory is *not* emptied when pressiong `C`
- Whenever the buffer is not empty, pressing `M` copies the content of the buffer to the memory.
- When the buffer is empty, pressing `M` should not change the content of the memory
- Pressing `MR` copies the content of the memory to the buffer. The value is *not* removed from the memory.
- It is not possible to press `MR` when the memory is empty

3.2 Exercise 1

3.2.1 Modeling

Write an EFSM to model the calculator according to the above specification. Each transition of your EFSM should model an action that can be implemented by calling one of the methods in the `Calculator` class, or a sequence of those methods.
You can design your EFSM on paper, or using a drawing program such as the drawing tools in Libre Office.
You may want to design more than one model, in order to test different aspects of the behaviour of the `Calculator` interface.

3.2.2 Implementation

Write the implementation of your EFSMs in Java as classes that inherit from `FsmModel`. Then, write a JUnit class that instantiates these EFSMs and the SUT, write an *adaptor* to connect the model with the SUT, and try out the available ModelJUnit traversal algorithms to generate tests from your models by using ModelJUnit. Report state and transition coverage for each of them.
When a test doesn't pass, identify if the problem is in the implementation or your model. In the last case, correct your model and rerun the tests, otherwise report on the implementation error. (You are not requested to correct the implementation errors, but if you do correct them you can rerun your test cases to be sure no new error were found.)
Write a very short summary on the evolution of your model design taking into account the errors you found while writing your model (found through testing).

3.3 Exercise 2

Extend your model with an error state such that you can generate test cases not only for the expected good behavior but also for the exceptional or bad ones. (Hint: your EFSM should be complete.)
Repeat the rest of tasks requested in exercise 1 (implementation and testing) using this new model.