ECSE 429 - FALL 2015
ASSIGNMENT

Group 05

Nicholas Colotouros (260531370)
Ze Qian Zhang (260483633)

How to run the test class generator

To generate the test class, put the xml file in the root of the project folder and run the JUnitTestGenerator.java file with the name of the XML file. Not providing a file name will result in the application using CCoinGenerator.xml as an input. The generated test class will be inserted into the test folder with the same package structure as the class it is generating and any existing tests of the same name will be overridden. If the file is not found the program will notify the user and terminate without generating anything.

Manual Changes

Though some basic string parsing, we were able to automate the generation of certain simple condition and action verification, namely for the following patterns (the spacing is very important, it is also critical that these variables have a corresponding getter in the implementation, which start with “get” for int_variable and “is” for boolean_variable):

1. boolean_variable = boolean_value;
2. int_variable =/</> int_value;

For any other pattern of condition or action, we must add the verification code manually, given that there is too much uncertainty in the variety of patterns, and it would require massive amount of work to recognize them all. Moreover, it is hard to know whether a variable is a public property or can only be obtained with a getter, whose name may vary wildly depending on the naming convention. For example, the following action verification had to be added manually:

1. int_variable = int_variable +/- int_value;

The above pattern requires the following code: we have to instantiate a corresponding variable at the beginning of the test; get the value of int_variable just before triggering the event and store it; then finally we have to get again and assert the value of int_variable with an assert statement after the event.

Lastly, we also had to manually add “SMachine.addQtr();” in multiple tests to satisfy the condition for curQtrs such as “curQtrs > 3”. It would require a lot of work to be able to recognize the additional events that we need to trigger in order to satisfy the condition for a certain transition. Thus, this code had to be manually inserted.

Defects found in CCoinBox Example

1. When the machine receives the event “addQtr” while in the state “allowed”, it would go to the state “notAllowed” mistakenly. This defect was fixed by modifying the argument passed to the method “setState” to “allowed” (line 142 of original CCoinBox.java).

2. When the machine is in the state “allowed”, if “returnQtrs” or “reset” is called, it does not set “allowVend” back to “false” correctly. This defect was fixed by adding the line “setAllowVend(false);” to both “returnQtrs” and “reset” in the switch statement for the state “allowed”.

The Main Challenges of Automating Sneak Path Test Cases

The main challenge of automating sneak path test cases would be dynamically setting up each possible state so that each possible action can be run on it while considering any potential conditions. For example, maybe testing for sneak path x when the finite state machine has a collection of 4 items won’t work, but the sneak path will work when it has a collection of 3 items.

Another potential difficulty is handling exceptions. What happens if testing for a sneak path results in an exception being thrown? It may be safe to conclude that the sneak path doesn’t exist in some cases but at the same time it ties into the idea of sneak paths only appearing in certain conditions.