ECSE 429 (Software Validation)
Assignment One Report

Geoffrey Long (*260403840*)
Kelly Morrison (*260459446*)

October 5, 2015

1 Running the Source Code

Import the ConformanceTest.zip file into Eclipse using the Import feature of Eclipse (File $\rightarrow$ Import $\rightarrow$ General $\rightarrow$ Existing Projects into Workspace $\rightarrow$ Select archive file). To generate tests for a given state machine (xml format), add the full path of the xml file to the argument of the project (Run $\rightarrow$ Run Configurations $\rightarrow$ Arguments $\rightarrow$ Program Arguments), then click "apply" then click "run". This will run the main class, ConformanceTest, which automatically generates the JUnit Test file, which is called Generate$< ClassName >$ where ClassName is the name of the class associated with the xml file. You will find this test class saved in the same package as the implementation of the state machine. Open this class and run it (Run $\rightarrow$ Run As $\rightarrow$ JUnit Test).

2 Discussion of Manual Changes

When we run our automatically generated test class on the CCoinBox state machine, we have 2/10 tests that fail. This is because certain conditions are not met. Specifically, when you are in the allowed state of the machine, there are three transitions that execute the vend event. The specific transition that is triggered depends on the value of curQtrs (the condition). Although conformance tests are generated for paths containing each of these transitions, the round trip path up to the preceeding state ("allowed"), and therefore the value of the condition (curQtrs), will be the same. In this case, the value of curQtrs is 2 for all three transitions. So the tests checking the other paths - one requiring curQtrs==3 and another curQtrs>3 - will always fail.

This issue was dealt with in the manual test class, TestCCoinBox, via manipulation of the condition variable. In this class, we added a while loop to increment the value of curQtrs until its value satisfies the condition for the transition being tested. After adding these manual changes, our test class passes for all ten conformance tests.

This shows that our automatically generated test class will not be able to handle cases where there are multiple transitions from a specific state with the same event where the event invoked depends on a condition variable. This has to do with how the round trip path tree is generated. All of the transitions out of a certain state will be added to the tree when the state is reached. This means that a path to a given non-leaf state will always be the same, and therefore the value of the condition variable will always be the same. This condition variable will have a specific value, and therefore will only satisfy one event.

Although it is possible to automatically generate code to ”force” the values to match the conditions, this might not be desirable. It is difficult to say if a condition is failing because it is supposed to fail, or because it simply didn't take the right path through the tree. Essentially, it would be hard to automatically generate a test class that takes care of these cases without knowing the nature of the state machine. You would also need the code to test the boundary values for these conditions. This is seen in testConformance_11() in TestCCoinBox.java. The generated code did not pick out the error in CCoinBox because the test passed when curQtrs was 4 (which satisfied the condition) but not when curQtrs was 5. Although we could add in tests which test different boundary conditions, due to the issues surrounding conditions discussed earlier, this is not necessarily desired behaviour and may not function properly in all state machine implementations.

It is worth noting that similar behavior can be seen in the automatically generated test class of the Legislative state machine. There are 2/6 tests that fail because the value of the variable isCommonBill does not match the expected value.

3 Discussion of CCoinBox Defects

In the addQtr() method of the CCoinBox.java class, there is an error in the case allowed section (line 142). According to the diagram on the PDF and the XML file, when you are in the allowed state, and you call the addQtr() method, the new state should be allowed, but in the java code, the new state is set to notAllowed. We updated the value of the new state to allowed to match the state machine from the XML file.

Similarly, in the reset() method and the returnQtrs() method of the CCoinBox.java class, there are errors in the allowed case. For both methods, when you are in the allowed state and the new state is empty, the allowVend Boolean should be set to false. This is again according to the XML file and the PDF. We added this statement to both the returnsQtrs() and reset() methods in the allowed cases (lines 113 and 175 respectively).

Finally, there was an error in the vend() method. In the case where getCurQtrs()>3, curQtrs should be set to curQtrs = curQtrs - 2 to match the given diagram. In the original method the code was simply resetting the value of curQtrs to 2 instead of decrementing by 2.

4 Generating Sneak Path Test Cases

As per the lecture slides, testing the round trip path tree demonstrates conformance to the ”explicitly modeled behaviour”. This explicitness allows automated test generation. Sneak paths, however, are implicitly excluded behaviours. These cannot be parsed from the information we are given.

Although we could automate a test generator that iterates through each state and forces it to each illegal action with each possible entry condition, this would be a very large number of test cases. In addition, it would be difficult to automatically define the appropriate action that should be taken in each of these unspecified actions. It is hard, if not impossible, to know what actions should be taken or what the end state should be in such a situation. An appropriate reaction to an unexpected event is not something we could handle through automation, although we could probably force this unexpected event to occur.