Advanced Model JUnit - CPS3233

Model Testing & Model JUnit Refresher

- You create a formalisation (model) of the system which is an abstraction of the system.
- Upon execution, different paths within the model will be traversed. For each transition (action) three steps occur;
    - Trigger SUT
    - Update model
    - Check equivalence between SUT and model
- Model JUnit automatically generates the test cases itself (unlike RV, where the user drives the SUT).
- Model JUnit has no inherent notion of state. The only state variable is generally instantiated and modified by the user. This means that Model JUnit may execute a transition with a source state which it is not currently in. We control this via guards.
- We aim to create a test suite which
    - Sound: If a test case fails then a bug exists
    - Complete: If there is a bug then a test case will fail

Coverage

- There are 5 implementations of the abstract Tester class within Model JUnit
    - Random - Random walks
        - Greedy - Gives preference to transitions that have never been taken before. Once all transitions out of a state have been taken, it behaves the same as a random walk.
        - Quick - Remembers and prioritises unexplored paths.
    - All Round - A GreedyTester that will terminate each test sequence after getLoopTolerance() visits to a state.
    - Lookahead - A test generator that looks N-levels ahead in the graph.
- There are 4 coverage metrics which can be defined
    - Action - No. of distinct actions.
    - State - No. of times each state has been entered.
    - Transition - No. of transitions.
    - Transition Pair - No. of distinct pair of transitions.
- By default Model JUnit does know the automata it is working with so we need to build the graph. We do this via the buildGraph() method executed on the created tester.
- You can save the graph by using the printGraphDot() method on the Graph Listener after building the graph. Specify the path to a .dot file as a parameter.

Probability & Randomness

- Randomness during testing can be beneficial since it allows elimination of test bias while testing a wider range of conditions i.e. rather than hardocing a specific ID we can generate a number between -100 and 100 and ensure that execution is valid for different numbers.
- The downside of randomness is that it does not necessarily create reproducible tests, i.e. we do not know the parameters used for the test to pass/fail which may cause flaky tests, i.e. tests which fail under a specific set of conditions. These flaky tests uncover a specific scenario where the test expectation does not match the actual result but we do not necessarily know the conditions to fail the test.

Data Generation

- In model testing there are 2 potential areas where randomness may be used;
    - Parameters - Methods which require parameters as an input may be randomised. For example in a lift system we can decide to go up one, two, three … floors, not necessarily always one.
        - This can be achieved by using the Random() method in Java or a library like Podam.
        - Alternatively, one might prefer to not have randomness and hardocode all parameters.
    - Path randomness - This is specific to Model testing. An execution does not necessarily need to go through the same path for every tests case. In fact randomness here might be preferred in order for the test to cover different paths for between executions.
        - This can be achieved by setting the random field of the tester to a different value every time via the .setRandom() method.
        - Alternatively, one might prefer to hardcode a value within this method which is the random seed. A seed is a value which when fed into a pseudorandom function will always generate the same pseudorandom output.

Varying Delays

- When the unguarded action is called (using a timed model) one may decide to wait/delay for a fixed amount or a random amount.
- This can be achieved via the Random() function in Java. Make sure that the time is fast forwarded by the same amount.

Resetting Tests

- During testing, Model JUnit may decide to reset the SUT along with the model side variables. This may be required in order to fully traverse the model in cases where a model splits, i.e. a specific state/transition can't be reached from the current state irrelevant of the path takes. Madel JUnit may decide to reset itself at random.

- Each time a reset is triggered the overriden reset method is executed. This method should reset all model parameters to their initial value including the state. The testing parameter dictates whether the the SUT should also be reset.
- The reset functionality is controlled by the tester and the default reset probability is set to 0.05.
- In order to override this value pass the required reset probability to the setResetProbability() method. The value should be between 0.0 and 1.0.

Timeout Probability

- This is a tied model attribute which controls the probability of a timeout being taken when one or more timeouts are enabled. The default value is 0.25.
- In order to override this value pass the required probability to the setTimeoutProbability() method. The value should be between 0.0 and 1.0.

Emulating a Markovian Model

- A Markovian Model is similar to a FSM but each transition has a probability assigned to it. It is important that when the probability of all outgoing transitions are added the result is 1.
- This can be achieved by generating a random number within the guard and making sure that the number satisfies that of the guard. This means that for a transition to execute it must be chosen randomly by Model JUnit and the generated random number must satisfy the chosen probability for that probability.
- An alternative approach includes generating/updating the random number and comparing it to the probabilities of each guard. This would reduce the number of updates to the random number but it is more prone to running the unguarded action should many output transitions with low probabilities exist from a specific state.
- In the case where all guards fail the unguarded transition will execute. In order to avoid this one can decide assign all probabilities a range within the valid values such that a transition will always execute. This approach requires the random number not be reset between checking the different guard conditions (i.e. the second approach mentioned above).

Time

- Until now we have not covered real time properties (i.e. properties which depend on time not just state). In order to do this we can use Model JUnit's timed automata.
- The first difference is that rather than implementing FsmModel you will need to implement TimedFsmModel. This will allow you to use Model JUnit's timing feature (an integer annotated with @Time). This integer will track the number of time increments which occur but it does not inherently represent seconds/milliseconds etc. How much 'real' one time increment represents is determined by the user.

- The TimedFsmModel requires the implementation of a new method named getNextTimeIncrement(). This method is executed after every transition and controls the value of the now variable. A user may decide to implement custom logic if the increase in time is not always uniform.
- Additionally when creating the tester you may opt to wrap the model within a TimedModel.
- Should you add time properties within the guards a situation may occur where you are not able to perform any transition even if you are at a correct state because the real time property is not satisfied. Remember that execution terminates if no action can be traversed and thus we need to implement an unguarded action.
    - As the name implies an unguarded action is an action without a guard. Typically the unguarded actions does not contain any logic or state changes but it is simply used to ‘pass the time’ (i.e. increment the time).
- Guards can also be added within assertions in order to check that real time properties are being followed.
- In order to track real time properties one can use one of the below approaches;
    - Timeouts
    - Thread.sleep() and/or while()

Timeouts

- This approach leverages the @Timeout annotation provided by Model JUnit in order to simulate (speed up) time.
- Variables annotated with @Timeout act as the guard for their respective action and execute only when the time is equal to the given value.
    - If there is no action is associated with the new time increment then the unguarded action will trigger.
    - If multiple timeouts match that time increment then the their execute is determined by the timeout probability (although I have not managed to make this work)
    - In order to switch off the timer set the value to -1.
- This approach is very useful since you are easily manipulating/fast-forwarding time because for each time increment you are not actually waiting the defined time slice (for example 100ms).
- The viability of this approach depends greatly on the ability to mock/manipulate time for the SUT. Since our model is moving faster than real time we need to alter the timing of the system in order to remain in sync. This implementation is dependant on the actual implementation (and testing requirements) of the system. One way this can be achieved is by mocking the time variable using libraries such as Mockito and Power Mock.

Thread.sleep() and/or while()

- This approach uses the Java Thread.sleep() and/or while() functionality in order to allow time to pass. This approach is more relevant for approaches where we want to

test real time properties but we do not have the possibility to mock the time of the SUT.
- For each time increment the model needs to wait/sleep the correct amount of time depending on the weight of each time slice in order to remain consistent with the SUT. Remember, the getNextTimeIncrement() method is called after each transition.
- Time checks need to be added within the guards to ensure that the system is allowed to traverse the selected condition. Therefore, it is important that an unguarded transition is present within your system for cases where the guard for all transitions fail.
- By default a transition will take the time defined within the getNextTimeIncrement() method but this does not necessarily have to be the case. You wait within a transition if required and increment the timer by the subsequent wait time.

Merging Both Approaches

- It is possible to merge both approaches together by using @Timeout model-side in order to complement the guards but still use sleep/while in order to stay synchronised with the SUT.

Notes

- If you have any problems refer to
    - Javadocs via your IDE or via grepcode;
        - grepcode.com/snapshot/repo1.maven.org/maven2/nz.ac.waikato.modeljunit/modeljunit/2.5/
    - These online tutorials;
        - cse.chalmers.se/edu/course/DAT261/tutorials/modeljunit.html
        - cse.chalmers.se/edu/year/2015/course/DAT260/labs/modeljunit-tutorial.html
    - Additional Reading: Practical Model-Based Testing (Utting and Legeard)
- View .dot files online: webgraphviz.com
- Questions regarding the assignment will only be answered through VLE. Any emails/facebook.
    - Hint: By default Model JUnit follows one FSM i.e. there is no nesting of objects (like Bank -> Users -> Accounts -> Transactions). How can we do this in Model JUnit?