CPS3230 – Fundamentals of Software Testing - LARVA Tutorial

Runtime Verification

- No matter how many tests we try out we still cannot predict all the possible circumstances under which the system will operate.
- Solution; we monitor the system at runtime.
- We define properties based on the program's execution (and therefore our model)
- If a violation is observed an alert can be triggered.
- System is considered as a **black box.**
- The system drives the monitors, not vice versa.

Installation

1. Download Eclipse for Java developers. It is recommended that you use Eclipse Luna (https://archive.eclipse.org/eclipse/downloads/drops4/R-4.4.1-201409250400/) since certain compatibility issues exist with newer versions of Eclipse. You need to have Java 8 (https://adoptium.net/temurin/releases/?version=8) as your default JDK for Eclipse Luna to work (not Java 9).
2. Go to Help > Install New Software > Add > Enter http://download.eclipse.org/tools/ajdt/44/dev/update/ in the location tab > Ok > tick *AspectJ Development Tools (Required)* > Next > Next > Accept the terms of the license agreement > Finish
3. Repeat the above but use http://www.cs.um.edu.mt/svrg/Tools/LARVA/update-site/ as the URL.
4. Create a new AspectJ project within your workspace by going to File > New > Project… > AspectJ > AspectJ Project. Name it BadLogin or BankSystem, depending on the code you are going to import into the project. Choose your JDK version and click finish.
5. Copy the contents of the src folder of the downloaded code into your src folder within your newly created project. You should have a package with the code and a .lrv file (.lrv file only for the BadLogin example)
6. Upon opening a file within your project you will be asked whether you want to add Xtext nature to your project. Select Yes.

Larva Syntax

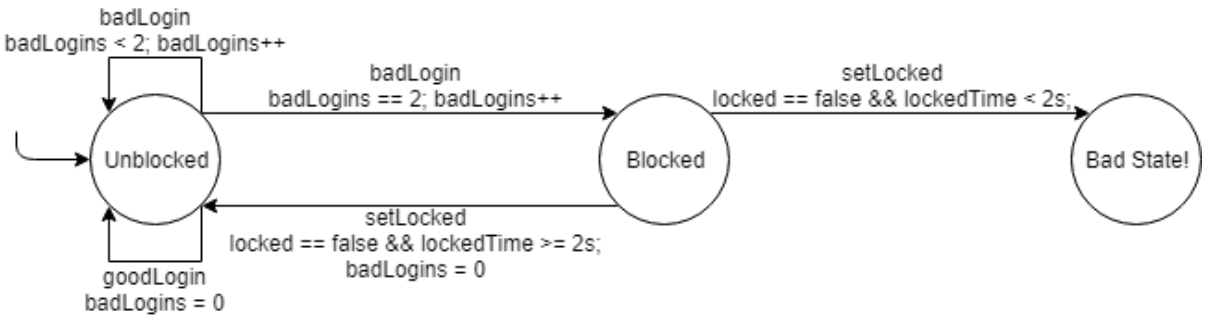
- **Global:** Outer container of the script. Contains all components except imports, java code etc.
- **Variables:** Creates (and initialises, if necessary) the variables required. Also, instead of creating them here you can initialise variables in the starting state.
- **Events:** Define the methods in the code which are important for us (generally because they signify a transition). You can bind variables which are not strictly related to the method call by using the where clause. When calling methods which have parameters which are not relevant, use a wild card (* or any string ex. arg1). Can also define a collection of events which will trigger if one event of the collection is triggered by enclosing event names in curly brackets. These events are triggered BEFORE the method executes. You can opt to execute this when the method returns (uponReturning), when an exception is thrown (uponThrowing) and when an exception is handled (uponHandling).

- **Properties:** Contains states and transitions. There can be more than one property and thus they need to be given a name.
- **States:** Contains states that will feature in your state machine, i.e. the ones you defined in your automata. There are four types; accepting (desirable for system to terminate in), bad (system should not be here), normal (not accepting and not bad) and starting (automata starts here, only one can exist here) written in this order! Can omit any state associations you don't want to use but you must always have the starting state. You can define an operation upon entry of the state.
- **Transitions:** Transitions similar to the ones you have in your automaton. Contains an event (name of event excluding parameters), condition (condition whether the transition can be executed) and an action (what to do once the transition happens). You can omit the condition and the action if necessary. Since more than one transition can exist from the same state, the code executes the first transition that satisfies the condition starting from the top! This means that you need to and conditions if necessary. No events triggered from the actions of the events.
- **Foreach:** Keeps track of multiple objects of the same type (i.e. one automaton per object). You need to know the construct of which object is being used so you specify the context so you only effect the automaton of the object being modified. To do this you use the where clause in events to make sure you are using the correct context. You can overwrite the equals and toString method if necessary. Foreach scopes can be nested (a foreach can contain a foreach) and but in the case of nested foreach you need to compare both the actual context and the parent context. You can refer to variables in the parent scope by using the :: notation.
- **Invariants:** Only found in foreach blocks and they make sure that a specific variable doesn't change value. Invariant checks can be enabled/disabled after a particular transition.
- **Imports:** One of the few blocks which is not in the global scope. Defines which packages need to be imported for the larva code to execute. Written above the global scope.
- **Methods:** You can define your own java methods in the LARVA script (written under the global scope). You do not need to specify an access modifier. Static methods can't access monitor variables. You can also import java code from a file but you need to define the imports within your script.
- **Clocks:** Can be used as variables and they are used to define real time properties. Defined in the variables scope and they can execute once after X amount of seconds using the @ notation or once every X seconds using the @% notation. The clock starts running on startup or upon the start of a new context. reset() will restart the clock from 0, current() will return a double with the elapsed seconds, compareTo() compares two clockes were 0 = equal, +ve = larger than parameter, -ve = smaller than parameter, off() turns off the clock, pause() pauses the clock, reset() resets the clock.
- **Channels:** Communication between automaton ex. start automaton upon completion of another. Visible to the current scope and lower and doesn't take care of contexts. Defined in the variables section. Use the send() and receive() methods to communicate between channels.
- **Comments:** Use %% for comments.

Notes: Use the documentation. It delivers good explanations coupled with exhaustive examples. Do not use variables which start with an _ (syntactically correct but generated code/files use the _ syntax) You can download a graphical package which shows your properties as images. The methods your LARVA code hooks into have an orange arrow on the left side of the Eclipse editor.

Question 1 – BadLogin

The system simulates a number of good and bad logins on an account. An account is locked for 2 seconds after 3 bad logins but then the user is allowed to log in again. The model is as follows;



Question 2 – BankSystem

You have been provided with an implementation for a bank system. This system is layered; the *system* has a number of *users*, each who own *accounts*, which in turn contain *transactions*. The bank system allows you to create, edit or delete users, accounts and transactions (ignore the process choice). When you edit an entity you would be presented with its children (i.e if you decide to edit a user you will be presented with a main menu where you can add, edit or delete accounts).

The system does not contain any business logic which checks your operations, so you are allowed to delete accounts which do not exist, add an infinite number of entities etc.

Your task is to create a model which shows whether the system has more than 5 users or more than 5 accounts per user or more than 5 transactions per account at any point in time. Once a model is defined implement this model as part of a runtime verification implementation using LARVA.