LarvaLight User Manual

LarvaLight is a simple tool enabling the user to succinctly specify monitors which trigger upon events of an underlying Java system, namely method calls and returns. If the events and their parameters lead the monitor to a bad state, then an assertion failure is triggered, alerting the user. The rest of this document explains how monitors can be specified in LarvaLight and how they can be used to assert a system's behaviour. **For a quick start see the last section.**

1. Setting up LarvaLight

To use LarvaLight, include the accompanying JAR file in the class path. Subsequently, the asserted Java system should be compiled and run using AspectJ. It is also crucial to include the JAR file in the aspect path and enable assertions. If you are running your system **outside of Eclipse**:

1. First compile your code (assuming a main class in a package called "ordinary") using AspectJ:

```
ajc -cp ".\aspectjrt.jar;.;.\larvalight.jar"
   -aspectpath ".\larvalight.jar" src\ordinary\Main.java
```

2. Secondly, run your application using AspectJ with assertions enabled:

```
call aj5 –ea -cp ".\larvalight.jar;.\src" ordinary.Main
```

In Eclipse (assuming AJDT plugin is installed):

1. First ensure that the project to be monitors has been converted to an AspectJ project. This can be done by: *Right-click Java project >> Configure >> Convert to AspectJ Project*
2. Next, setting the class path can be done by:
    a. *Paste* `larvalight.jar` *under the project folder and refresh the project in Eclipse*
    b. *Right-click Java project >> Build path >> Configure build path...*
    c. *Go to the Libraries tab >> Add JARs... | Choose* `larvalight.jar`
3. The aspect path parameter can be set through the following steps: *Right-click Java project >> AspectJ Tools >> Configure AspectJ Build Path | Choose* `larvalight.jar`
4. Finally, it important to ensure that Eclipse is using a JDK to run the monitored system and that assertions are enabled:
    a. *Go to Window >> Preferences*
    b. *From below Java choose Installed JREs*
    c. *Select a JDK (if no JDK is available add one by choosing Add...) and click Edit...*
    d. *In the "Default VM arguments" add -ea*

2. Specifying events in LarvaLight

In order for LarvaLight to perform monitoring, it needs to hook on to system events. LarvaLight support method call-based events, i.e. events which trigger upon a method call. Thus, specifying an event involves specifying a method signature or part thereof which will be used to trigger monitoring activities. In this section we describe in detail how events can be specified in LarvaLight in terms of four components: the target class, the method name, and a parameter pattern. For exemplification purposes we will define events with respect to a non-static method call `deleteAccount` within a class `Bank` which takes a parameter of type `Account` and a non-static method call `getBalance` within a class `Account`.

1. **Position** – The position specifies whether the event triggers before, after, or over an exception throw of a method call. Thus, for example the event `before deleteAccount` triggers before the execution of the `deleteAccount` method, `after deleteAccount` triggers when `deleteAccount` successfully returns, while `onthrow deleteAccount` triggers when `deleteAccount` terminates with an exception throw.
2. **Target** – The target of the method is optional but can be used to specify the class name of the method call (to narrow pattern matching). Referring to the example, `Bank.deleteAccount`, `B*.deleteAccount`, and `*.deleteAccount`, all refer to exactly the same event.
3. **Method** – The method name can also be left unspecified or pattern-matched using the * wildcard. Therefore `Bank.*`, `Bank.`, `Bank.delete*`, `*.delete*`, and `delete*`, all refer to the same event. Note that the method name `new` can be used to refer to the constructor.
4. **Parameters** – Parameters can be used to pattern match the method signature if the method is overloaded. For example `deleteAccount(Account)` ensures that only methods which take a single parameter of type `Account` are matched.

Apart from specifying the method call to be matched, events can be used to provide values to variables by binding variable names to different parts of the method call. There are three ways in which this can be done:

1. **Return value** – The return value of a method (not of a constructor) can be bound to a variable (e.g.: `balance`) as follows: `balance = Account.getBalance`. Note that `before balance = Account.getBalance` is invalid since the return value of the method would not be available before the method's execution. Moreover, if the event triggers over an exception throw, then the returned object must be of `Throwable` type.
2. **Target** – The target object of the method call can also be bound to a variable as follows: `acc.getBalance`, signifying that the `Account` object on which `getBalance` is called is bound to variable `acc`. Note that pattern matching the object returned by a constructor call should be treated as a target binding e.g. `acc.new()`.
3. **Parameters** – The parameter pattern can also be used to bind variables to values. For example `deleteAccount(acc)` binds the parameter of the `deleteAccount` method call to variable `acc`.

3. Specifying a simple monitor in LarvaLight

The most basic monitor is LarvaLight can be specified in terms of a number of rules composed of an event, a condition, and an action. An example of a rule would be:

```
withdraw(amount)\\amount<=1000
           -> System.out.println(\"Withdraw limit ok: \"+ amount);
```

This rule triggers if the event `withdraw` occurs and the condition `amount<1000` is satisfied, causing the executing of the action command. If the triggering of the rule signifies that the monitor has detected unexpected behaviour then `-X` is used instead of `->` :

```
withdraw(amount)\\amount>1000
      -X System.out.println(\"Withdraw limit exceeded: \"+ amount);
```

Note that all the components of the rule can be omitted: if no event is specified then the rule will trigger on any event; if no condition is specified the rule will always trigger upon the given event; while if no action is specified, the rule will not execute any code. Finally, apart from rules, a monitor requires a label to distinguish it from other monitors, and a list of the variables bound through the events or used in the rule conditions and/or actions. In the example of checking the withdraw amount limit, the label is `withdrawLimit` while the only used variable is `amount`. The specification would thus be as follows:

```
Rule.create("withdrawLimit",
      new String[] {"double amount"},
      new String[] {"withdraw(amount)\\amount>1000 -X
      System.out.println(\"Withdraw limit exceeded: \"+ amount);"});
```

4. Specifying more complex monitors in LarvaLight

In more complex monitors, one might need to import classes which are mentioned in the rules. For example if the Account class is mentioned, then the class has to be imported by passing the parameter `import tutorial._6_rules.accounting.Account;` Note that several imports can be added either through the use of the * wildcard or by adding further `import ...` to the parameter. Furthermore, to enable the monitoring of several accounts at a time, a special variable parameter is specified to indicate that its value is to be used to distinguish between different instances of the monitor. Finally, the transition which signals that the current monitoring instance has completed successfully is signified by `-|`. To illustrate these features, the following example maintains the expected balance of the account so that it is compared to the actual account balance to detect any mismatch:

```
Rule.create("balanceCheck",
            "import tutorial._6_rules.accounting.Account;",
            "Account acc",
new String[] {"double amount", "double balance"},
new String[] {
      "acc.new() -> balance=0;",
      "before acc.withdraw(amount)\\acc.getBalance()!=balance -X
            System.out.println(\"Balance mismatch found\");",
      "before acc.deposit(amount)\\acc.getBalance()!=balance -X
            System.out.println(\"Balance mismatch found\");",
      "before acc.withdraw(amount)\\acc.getBalance()==balance ->
            balance-=amount;",
      "before acc.deposit(amount)\\acc.getBalance()==balance ->
            balance+=amount;",
      "deleteAccount(acc) -|"});
```

Note that the order of the rules is important as all the matching rules are triggered. Thus, for example updating the balance before checking for a mismatch will always result in a mismatch. To avoid such ordering issues, can use `-/` instead of `->` to signify that no further rules trigger. Consequently, the rules above can be written as:

```
      "acc.new() -/ balance=0;",
      "before acc.withdraw(amount)\\acc.getBalance()==balance -/
            balance-=amount;",
      "before acc.deposit(amount)\\acc.getBalance()==balance -/
      balance+=amount;",
      "before acc.withdraw(amount)\\acc.getBalance()!=balance -X
            System.out.println(\"Balance mismatch found\");",
      "before acc.deposit(amount)\\acc.getBalance()!=balance -X
            System.out.println(\"Balance mismatch found\");",

"deleteAccount(acc) -|"
```

Note that both `-X` and `-|` also block the triggering of any other rule.

5. Specifying a finite state machine

To facilitate the specification of properties with a particular pattern, LarvaLight provides a means of directly specifying finite state machines. Similar to the basic monitor described above, a finite state machine can be specified by calling the method `create(String label, String imports, String foreach, String[] transitions)` where transitions take the form of `source [event\\condition>>action] destination`. The starting state is always the state `start` while to signify a bad state, the state `bad` should be used and `end` should be used to signify the correct termination of a monitor.

Example

Consider for example a finite state machine which specifies the correct life cycle of an account object. The transitions specify the following rules: the account can be created and deleted (the first two transitions) in that order. However, no operations can be carried out on an account before it is created (next three transitions). Finally, it must be ensured that if an account has been created it is not created again (last transition).

```
FSM fsm = FSM.create("accountLifeCycle",
              "import tutorial._9_fsm.accounting.*;",
              "Account acc",
              "start     [acc.new()]                new",
              "new       [deleteAccount(acc)]       end",
              "start     [deleteAccount(acc)]       bad",
              "start     [acc.deposit]              bad",
              "start     [acc.withdraw]             bad",
              "new       [acc.new()]                bad");
```

6. Specifying a regular expression

To further facilitate the specification of monitors, the user can also use regular expressions. Each character in the regular expression is defined in terms of an event and the regular expression is defined in terms of the characters. Importantly, a regular expression can be used to either match expected behaviour (if a deviation is observed, a bad state is reached), or it can match unexpected behaviour (if the regular expression is matched, a bad state is reached). The rest of the parameters are described above; an example is given below.

Example

Specifying the account life cycle described above in terms of a finite state machine can be more succinctly specified as a regular expression as shown below:

```
RE.create("regular_expressions",
          "import tutorial._6_rules.accounting.*;",
          "Account acc",
          RE.Matching.EXPECTED_BEHAVIOUR,
          new String[] {
          "n = acc.new()",
          "x = deleteAccount(acc)",
          "d = acc.deposit",
          "w = acc.withdraw"},
          "n(d|w)*x");
```

7. Using a monitor

A created monitor can be started by invoking method `start`. The monitor can then be paused or reset by called the methods `pause` and `reset` respectively. Note that pausing the monitor temporarily stops event matching but keeps the monitoring state intact. On the other hand, resetting the monitor discards all the monitoring state but does not stop the matching. If one wants to both pause and reset the monitor, then one should call the `stop` method. Finally, a `dispose` method is available to remove all information about the monitor from the monitoring runtime environment. Any method invocation on a monitor after disposing of it would return false.

To avoid having to keep a global variable with reference to the monitor object, a static version of these methods is available and can be used by passing the name of the monitor as a parameter. Finally, the user can choose whether the monitor outputs log information regarding the activities being carried out in the background. To this end the static method `setVerbose` should be used.

Example

```
FSM.setVerbose(true);
FSM fsm = FSM.create("accountLifeCycle",...);
fsm.start();
...
fsm.pause();
...
fsm.reset();
...
FSM.dispose("accountLifeCycle");
```

8. Combining Monitors with JUnit

LarvaLight provides specialised support for integration with JUnit. This can be achieved in the following steps (NB: ensure JUnit is also running using a JDK rather than a JRE!):

1. **Declare monitors** – Using the standard JUnit `@BeforeClass` annotation, declare any monitors that will be needed throughout the test class. For example:

```
@BeforeClass
public static void init(){
   FSM.create("a monitor",new String[]{"start [event] bad"}); }
```

2. **Declare rule** – Using the `@Rule` annotation instantiate a `MonitorRule` instance. This will enable the monitor to intercept tests and apply the appropriate monitors to them. Note that the monitor rule will automatically take care of the starting, resetting, and stopping of the monitors declared through the monitor annotation (see further below). Without the rule declaration, the monitor annotations do not work.

```
@Rule
public MonitorRule mr = new MonitorRule();
```

3. **Annotate tests with monitors** – Any test which is to be monitored needs to be annotated with `@Monitor` which takes as input an array of strings representing the names of the monitors applied to that particular test.

```
@Test
@Monitor({"a monitor", "another monitor"})
public void test() throws Exception {
   //some code ... }
```

4. **Dispose monitors** – To ensure that monitoring resources are freed up, the `@AfterClass` annotation should be used to dispose of the monitors.

```
@AfterClass
public static void dispose(){
   FSM.dispose("testexception"); }
```

9. Grouping Monitors

For convenience, monitors can be grouped so that they can be started, stopped, reset, and disposed of through a single method call. This feature is provided through the `Oracle` class which in its constructor accepts any number of monitor declarations. Subsequently, any method invoked on the constructed oracle object is relayed to each monitor included in the constructor. The use of oracles to group monitors is particularly useful when LarvaLight is used in conjunction with JUnit as shown in the example below.

Example

```
@BeforeClass
public static void setup(){
      oracle = new Oracle(FSM.create(…),Rule.create(…));
      oracle.start();
}

@Before
public void before() {
      oracle.reset();
}

@AfterClass
public static void teardown() {
      oracle.dispose();
}
```

10. Timer API

To make it easy for the user to specify properties related to time, LarvaLight provides a Timer API with the following functionality:

- *[Constructor] Timer (String identifier, Long… firingTime)* – The constructor allows the user to give an identifier to the timer, and specify any number of firing times (in milliseconds), i.e. points in time the timer would fire each time it is reset. Note that the identifier is optional and the constructor does not reset the timer automatically. In other words no alarms are set upon construction.
- *reset ()* – Each time the timer is reset, any firing times from the previous reset are discarded and a new alarm is set for each firing time declared in the constructor. Note that this function automatically enables and unpauses the timer.
- *pause () / resume ()* – The timer may be paused or resumed.
- *enable () / disable ()* – The timer can be disabled so that any future alarms set during a reset are ignored (however if re-enabled any remaining alarms will still trigger). Note that upon construction and reset the timer is enabled and unpaused.
- *time ()* – Returns the duration of time elapsed since the timer's most recent reset in milliseconds. If the timer is disabled, the function returns zero, while if paused, it returns the time elapsed from reset till pause (excluding any duration the timer has been previously paused).
- *fire (Long millis)* – This method cannot be invoked by can be declared as an event in a monitor to detect timer alarms.

Note that methods except *time()* and *fire()* return the timer object itself to enable method chaining.

Example

In the example below, a timer is used to measure the time a user is inactive during a session (after login): The timer is initialised to trigger after 5 seconds, upon each activity the timer is reset and disabled if the user logs out. If at any point, the alarm goes off; it signifies that the user has spent 5 seconds without performing any activity after login.

```
Rule.create("inactive_for_5_seconds",
      new String[] { "Timer t=new Timer(5000l);" },
      new String[] {"login -/ t.reset();",
                    "deposit -/ t.reset();",
                    "withdraw -/ t.reset();",
                    "logout -/ t.disable();",
                    "t.fire -X System.out.println(\"Session has been
inactive for at least 5s\");"})
```

Quick Start 1 – Declaring Monitors

An example with basic rules

An identifier for the monitor

Declaration of any variables used

A rule: Event \\ Condition –X Action

–X signifies that if the rule fires, the assertion has failed

```
Rule.create("withdrawLimit",
      new String[] {"double amount"},
      new String[] {"withdraw(amount)\\amount>1000 -X
      System.out.println(\"Withdraw limit exceeded: \"+ amount);"});
```

An example with finite state machines

Import any relevant classes

For each unique instance of this variable, a copy of this assertion is spawned

start always signifies the initial state

```
FSM.create("accountLifeCycle",
      "import tutorial._9_fsm.accounting.*;",
      "Account acc",
      "start     [acc.new()]                  new",
      "new       [deleteAccount(acc)]         end",
      "start     [deleteAccount(acc)]         bad",
      "start     [acc.deposit]                bad",
      "start     [acc.withdraw]               bad",
      "new       [acc.new()]                  bad");
```

A transition (similar to a rule): Event \\ Condition >> Action [this example shows only events]

bad signifies the assertion has failed

An example with regular expression

```
RE.create("regular_expressions",
      "import tutorial._6_rules.accounting.*;",
      "Account acc",
      RE.Matching.EXPECTED_BEHAVIOUR,
      new String[] {
      "n = acc.new()",
      "x = deleteAccount(acc)",
      "d = acc.deposit",
      "w = acc.withdraw"},
      "n(d|w)*x");
```

The regular expression can either be to match correct or incorrect behaviour

Symbol definition in terms of events

The regular expression itself

Quick Start 2 – Using Monitors in Code

An example main method

This is optional but would help a new user understand what is happening

Declare any monitors you wish to use

```
public static void main(String [] args)
{

      FSM.setVerbose(true);
      FSM.create("accountLifeCycle",
                 "import tutorial._5$mon_trace.accounting.*;",
                 "Account acc",
                 new String[] {
                 "start    [acc.new()]                new",
                 "start    [deleteAccount(acc)]       bad",
                 "start    [acc.deposit]              bad",
                 "start    [acc.withdraw(double)]     bad",
                 "new      [deleteAccount(acc)]       end",
                 "new      [acc.new()]                bad"}
          ).start();
```

Don't forget to start your monitor!

```
      Bank b = new Bank();
      b.useBank();
```

Normal system code

```
      FSM.dispose("accountLifeCycle");

}
```

Dispose of your monitor when done