Runtime verification is the idea of adding monitors to a system which make sure that the system is being executed correctly. Gives you coverage during the execution of the program. System is considered as a **black box** and a model is generated which defines the execution of the system. The model should never end in a bad state!!

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 <u>where</u> 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 (<u>uponReturning</u>), when an exception is thrown (<u>uponThrowing</u>) and when an exception is handled (<u>uponHandling</u>).

Properties: Contains <u>states</u> and <u>transitions</u>. 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; <u>accepting</u> (desirable for system to terminate in), <u>bad</u> (system should not be here), (system should not be here), <u>normal</u> (not accepting and not bad) and <u>starting</u> (automata starts here, only one can exist here) <u>written in this order</u>! 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 <u>event</u> (name of event excluding parameters), <u>condition</u> (condition whether the transition can be executed) and an <u>action</u> (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 <u>where</u> clause in events to make sure you are using the correct context. You can overwrite the <u>equals</u> and <u>toString</u> 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. So 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.