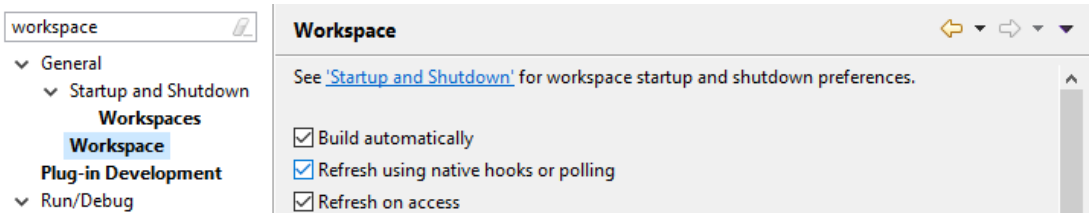
User Guide and Tutorial for the
Gamma Statechart Composition Framework

1. Installation

Gamma is tested to work with Eclipse 2021-12. Start with a new *Eclipse IDE for Java and DSL Developers* package. Install VIATRA (version 2.6.0) and the Yakindu Statechart Tools (version 3.5.13). Exit Eclipse and extract the Gamma zip file into the root folder of eclipse. Ideally this will create the *plugins* directory in the *dropins* folder of the root folder of eclipse, containing the JAR files of Gamma. (If not, make sure you copy all the JAR files contained in the Gamma zip file in the *plugins* directory of the *dropins* folder of the root folder of eclipse.) When starting Eclipse for the first time, you might need to start it with the *-clean* flag. Check if the plugin installed successfully in *Help > About Eclipse* and by clicking *Installation Details*. On the *Plug-ins* tab, sort the entries by *Plugin-in Id* and look for entries starting with *hu.bme.mit.gamma*.

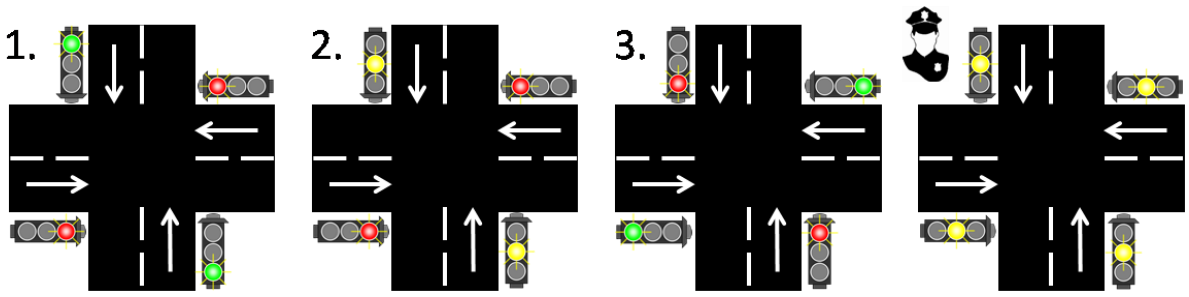
Tip: It is advised to turn on automatic refreshing for the workspace. The other option is to refresh it manually with F5 after every Gamma command.



For formal verification, download and extract UPPAAL. In order to let Gamma find the UPPAAL executables, add the *bin-Win32* or *bin-Linux* folder to the path environment variable (depending on the operating system being used).

2. Presenting the Models

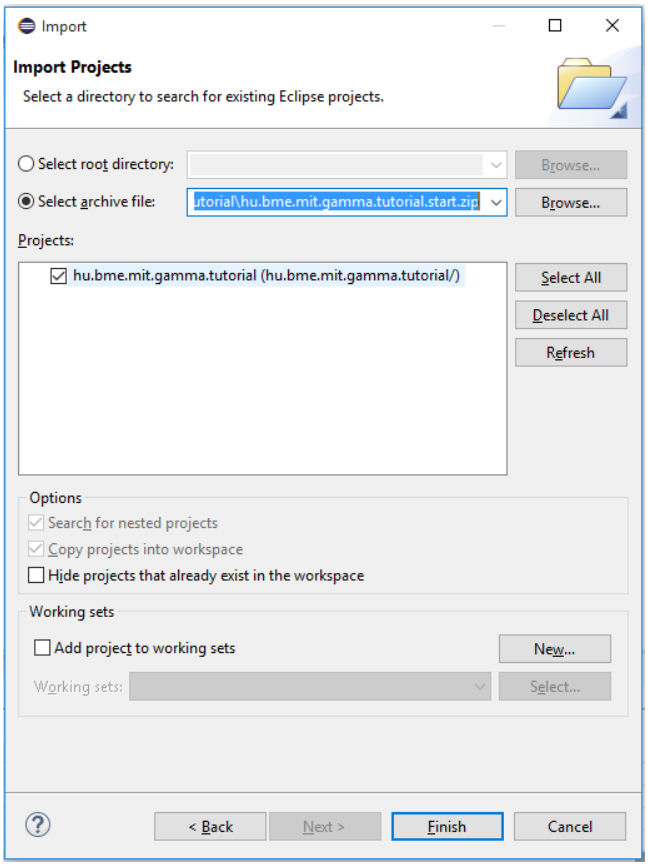
In this tutorial, we will design the controller of traffic lights in a crossroad. In each direction, the traffic lights are the standard 3-phase lights looping through the red-green-yellow-red sequence. As an extra, there is an interrupted mode that may be triggered by the police – in this state, the traffic lights will be blinking in yellow.

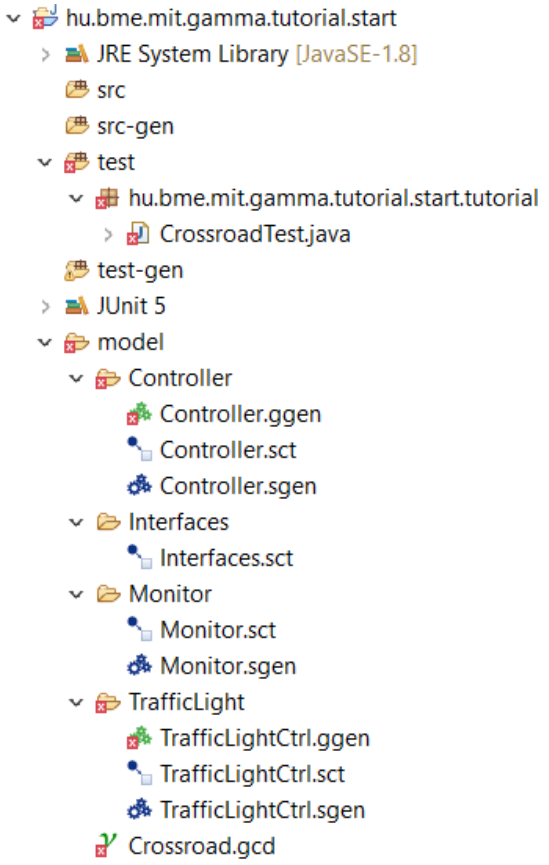


Import the skeleton of the crossroad model from *hu.bme.mit.gamma.tutorial.start.zip* as an existing project (browse the archive file then click *Finish*). You will see a number of existing artifacts, including a JUnit test file in the */test* folder and various models in subfolders of */model*. At this point, the project should contain errors, but we will fix them during the tutorial.

To reduce the complexity of the models, we divide the controller into submodules. For each road, the lights will be controlled by an instance of the *traffic light controller* statechart (*/model/TrafficLight/TrafficLightCtrl.sct*), while a separate *crossroad controller* (*/model/Controller/Controller.sct*) will be responsible for the coordination of the flow of traffic.

The models of the controllers should be easy to read. The *traffic light controllers* start from the *Red* state and will advance to the next state upon receiving a *toggle* signal. In this example, we assume that timing comes from the *crossroad controller* – in the form of such *toggle* signals. The *crossroad controller* will react to the passing of time, so that it can decide which traffic light(s) to toggle in the given step. This strategy separates the responsibility of handling the lights (through the *LightCommands* interface) and coordinating the flow of traffic.





As mentioned before, the police may interrupt the behavior of the crossroad at any time, switching all the lights to a blinking yellow state. This signal is sent through the *crossroad controller*, which will forward it to the traffic light controllers (as the blinking yellow behavior is implemented there).

There is also a *monitor component* (*/model/Monitor/Monitor.sct*) that will be used later.

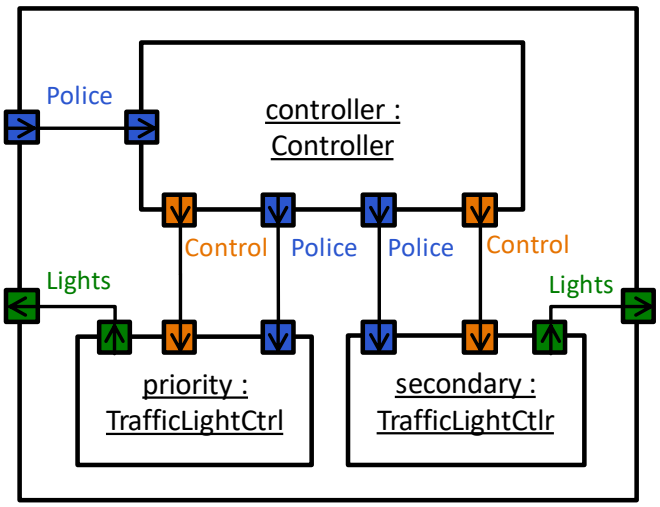
After examining the *traffic light controller* and the *crossroad controller*, you should notice that they have matching interfaces, but the direction of events is the opposite. This is because Gamma works with the concept of *ports*, points of service that can *provide* or *require* an interface. An output event on a provided interface will be an output event on a required one, enabling the connection of the two ports with a *channel*.

The interfaces used in the Yakindu statecharts of the controllers are defined separately. Ideally, the definition of the interfaces should be the first step in system design. Gamma supports this idea by

letting us define the interfaces in an empty Yakindu statechart, then compiling them into the native modeling language of the tool.

With the interfaces defined and components modeled, the last thing is to describe how the whole system is built. This is done in the textual syntax of Gamma. Open the composite system description in */model/Crossroad.gcd* (the extension stands for *Gamma Composite Definition*). You should see the skeleton of a *synchronous composite component*, which you will have to fill in.

To interpret the syntax, observe the following figure, which illustrates the schematic structure of the system. After importing the components, the file declares that we are specifying the *Crossroad* system, which will consist of a *CrossroadComponent* defined as follows.



- First, we define the ports of the system. In this case, we wish to *send* police interrupt signals from the environment (we *require* someone who implements this interface) and observe the output of the lights of the primary road (we *provide* the opportunity to observe the lights).
- Then we define the structure of the composite component in three parts:
  - we instantiate components with the following syntax:
    component *componentID : ComponentType*
  - define which port of which component should implement the ports of the composite component with the following syntax:
    bind *systemPortID -> componentID.componentPortID*
  - connect the ports of subcomponents with channels with the following syntax:
    channel *[componentID.componentPortID] -o)- [componentID.componentPortID]*.

Compositional Semantics

In this tutorial the synchronous-reactive semantics is utilized, which means that components are executed in cycles (just like the default behavior of Yakindu statecharts), all at the same time. In practice, the order of execution in each cycle is undefined, but this is not a problem, because communication over channels – the only legal way of communication in Gamma – is delayed by one cycle. This way, the causal relationship between the components is well-defined.
It is also important to note that Gamma considers the pieces of information passed through channels as signals (or events). In contrast to messages, these signals are synchronous, not queued, not buffered, they have to be processed in the cycle they arrive. One of the consequences is the restriction on multiple source ports for a channel – there is no way in synchronous-reactive semantics to distinguish the source and the signals will overwrite themselves in an undefined order.
Additionally, Gamma supports cascade (also in the synchronous domain) and asynchronous-reactive (based on messages and message queues) composition as well, not presented in this tutorial.

Note: By default, channels are 1-to-1 connections and no port can connect through more than one channel. The only exception is ports that implement a broadcast interface, an interface which has only outgoing events, in provided mode. Such ports may be connected to multiple listeners, ports that implement the broadcast interface in required mode.

3. Compiling the Yakindu Statecharts

Yakindu serves as a frontend to the formal modeling language of Gamma. Therefore, Yakindu statecharts have to be *compiled*. To compile a Yakindu statechart, Gamma first needs the definition of interfaces used in the system.

To generate the interface definitions from the existing empty Yakindu statechart, right-click */model/Interfaces/Interfaces.sct* and select *Gamma Commands > Compile Interface*. This should generate a new file called *Interfaces.gcd*. Press *F5* if you do not see the new file or turn on automatic refreshing of the workspace.

To compile the Yakindu statecharts modeling the two types of controllers, Gamma needs to know how to interpret the interfaces found in them. To specify this, we will use Gamma generator models (*.ggen*).

Open */model/Controller/Controller.ggen* to see what a Gamma generator model does. You will see that the file specifies the Yakindu statechart to map (this is the *name* of the statechart, which is by default the same as the filename, but can be changed in the properties view), then a series of mappings. For each interface in the Yakindu model, we have to create a port with an interface matching the Yakindu interface *in the specified mode* (provided or required). In this file, there are two pairs of ports that implement the same interface in the same mode, as well as one that implements the same but in a different mode.

With the specified information, Gamma can now compile the Yakindu statecharts.

Right-click the *.ggen* files one by one and select *Gamma Commands > Generate Artefacts*. This should create a new file for every statechart with the *.gcd* extension. This is the textual representation of statecharts used internally by Gamma.

4. Creating the Composite Model

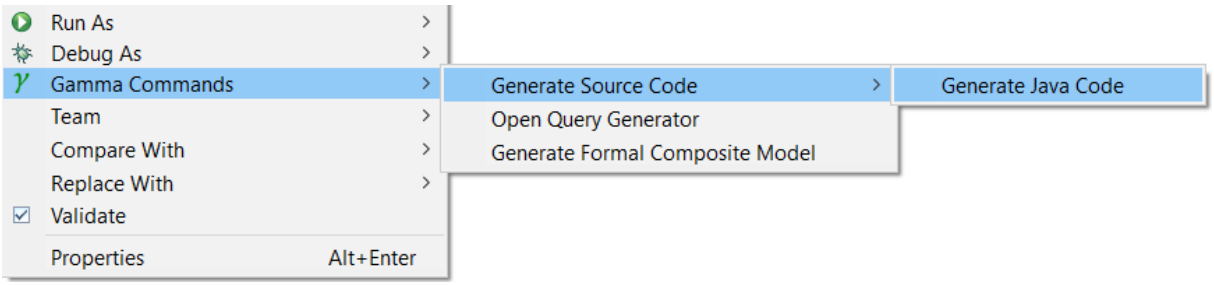
Finish the *CrossroadComponent* model in file *Crossroad.gcd* in accordance with the descriptions and figure presented above. Content assist can be used while editing the model by pressing *Ctrl + Space* on the keyboard.

By this time, none of the files in the */model* folder should have any error markers.

5. Code Generation

Gamma can generate source code for the composite system specified so far. It will reuse the code generated by Yakindu, so let us first generate the implementation of the Yakindu statecharts.

The project already contains the Yakindu generator models (*.sgen*) necessary to generate code from Yakindu statecharts. Editing and saving the statecharts will automatically regenerate the code, but we can also invoke code generation by right-clicking the *.sgen* files and selecting *Generate Code Artifacts*.



When Yakindu has finished code generation, let us generate the implementation of the composite system. Right-click on */model/Crossroad.gcd* and select *Gamma Commands > Generate Source Code > Generate Java Code*.

Note: C/C++ code generation is currently under development and therefore it is not yet supported.

After building the workspace, the last errors should vanish and the implementation of the crossroad should be in the */src-gen* folder in various packages.

6. Testing

To demonstrate the API of the generated code, there is a prepared test file in the */test* folder.

The file contains an embedded class (*CommandListener*) which implements a version of the *LightCommands* interface. Expand the imports to see exactly what is implemented:

hu.bme.mit.gamma.impl.interfaces.LightCommandsInterface.Listener.Provided

This Java interface is a listener for the output events of the *LightCommands* interface in *provided* mode. Other Java interfaces related to the *LightCommands* interface are as follows.

- *LightCommandsInterface.Listener.Required*: a listener for the output events of the interface in *required* mode.
- *LightCommandsInterface.Provided*: contains method to raise input events of the interface in *provided* mode.
- *LightCommandsInterface.Required*: contains method to raise input events of the interface in *required* mode.

The *CommandListener* will be used to cache the output events of the system and check them in assertions.

The file also contains an initializer method (*init*), which demonstrates how to instantiate the composite component. Instantiation also initializes the component, but it can be reinitialized if we need to register listeners before starting it.

There are two test cases in the file. The first one (*greenAtStart*) checks if the components initialize correctly: at first, the priority *traffic light controller* should emit a *displayRed* signal, while the *crossroad controller* will send it a *toggle* signal. This signal arrives in the next cycle, when the priority *traffic light controller* should raise the *displayGreen* signal.

Observe the comments to learn what the different methods do. Observe how to reach ports, how to raise events on them, how to register listeners and how to run the component until no more internal signals are left to process. There is also a *runCycle()* function that executes only a single cycle of the system, but that will leave internal signals in the channels.

The second test case is somewhat more complex and demonstrates the timed behavior of the implementation. Notice that we call the *reset()* method after registering the listener to reinitialize the component and therefore receive the first entry events as well.

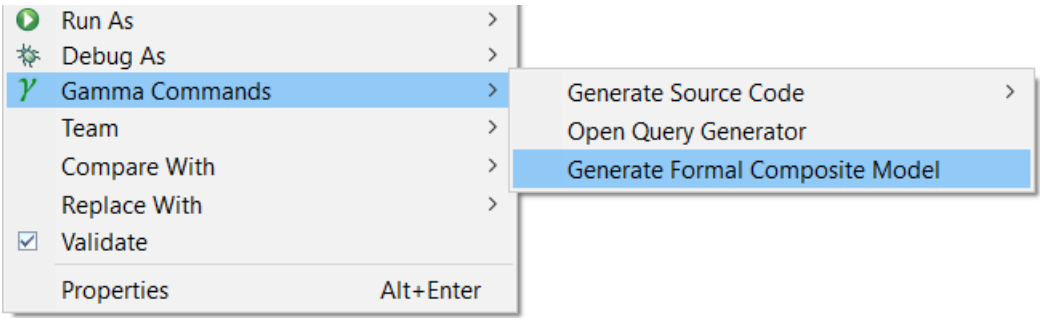
Run the tests by right-clicking the source file and selecting *Run As > Junit Test*. Both tests should be green, but the second one will take more than 3 seconds to run. This is due to the waiting in the test case, which is not scalable to larger tests. The next section will present a way to overcome this problem.

Note: If the second test fails, try to re-run it a few times. As system timing is not always accurate, there might be more extreme cases when the timers will not trigger when they would normally have to.

7. Model Checking

It is time for a deeper analysis of the crossroad model. Gamma can use model checkers to analyze the behavior of composite systems. Currently, the timed model checker UPPAAL is supported as a verification backend.

We have to start by generating the formal model that will be the input of UPPAAL. To do this, right-click on */model/Crossroad.gcd* and select *Gamma Commands > Generate Formal Composite Model*. This should create two more files, *Crossroad.xml*, which contains a model that can be opened with UPPAAL, and *Crossroad.q*, which contains a number of *queries*.



For each state in each statechart component, Gamma generates a query that will check if that state is active in any reachable configuration of the system. The queries can be checked in UPPAAL, either as a form of "deep validation", or to obtain a set of test cases. This functionality is reachable via the *Generate Test Set* button of the GUI.

Model checking of requirements, on the other hand, is an automatic formal verification technique used to exhaustively analyze the possible behaviors of a system and see if a desired behavior is possible to achieve, or a bad behavior can never be executed. We will use this capability to check if the crossroad can get into a state where both directions get a green light.

To specify the requirement, we need to use *temporal logics*. As temporal logic expressions are hard to read and even harder to write, Gamma provides a more intuitive way of formalizing requirements: fillable patterns.

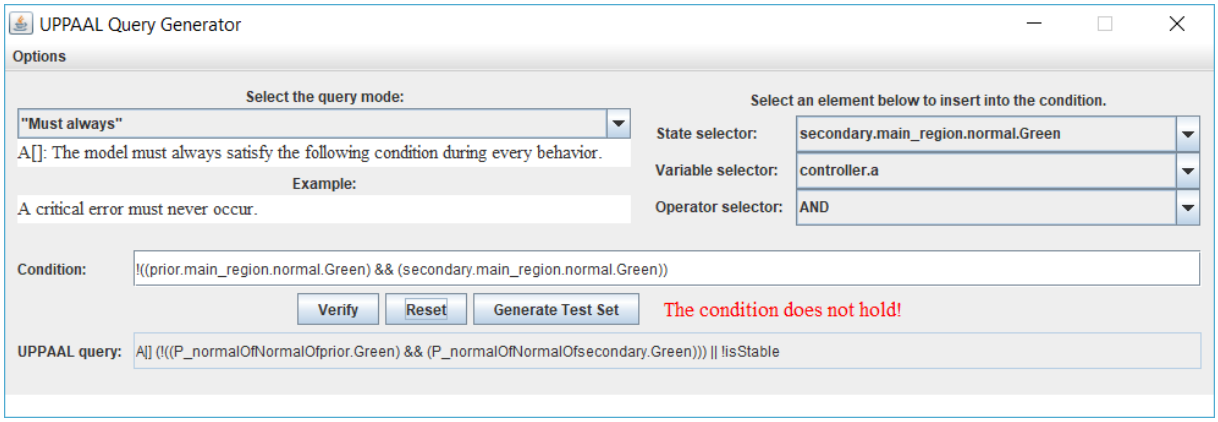
Right-click */model/Crossroad.gcd* and select *Gamma Commands > Open Query Generator* to open the requirement specification window. The top-left part of the window will let you select a template. There are five types of templates: *might eventually*, *must eventually*, *might always*, *must always* and *leads to*. Upon selecting one, the textbox below will show the corresponding temporal logic operator and an English sentence describing the requirement. There is also an example requirement that would be typically formalized this way. The middle part contains one or two textboxes to formalize the condition mentioned in the patterns. The top-right part helps in assembling the condition formula: select

something from the drop-down to insert it at the end of the textbox that last had focus. Note that the textboxes are editable, and the user has to take case of parentheses.

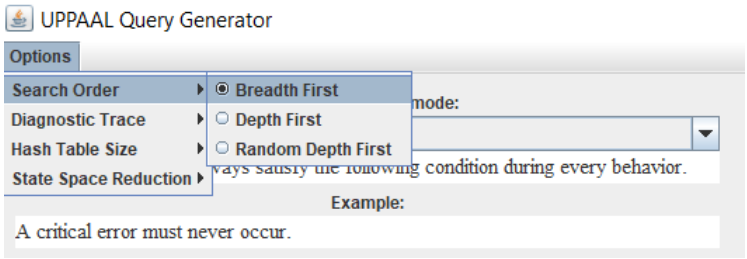
The bottom part contains the *Verify* button – click it after filling the conditions and UPPAAL will check if the model satisfies your requirement or not. Be aware that model checking is performance-intensive, so this operation might take long.

Let us specify the requirement of not having green light in both directions. Using the presented controls, select the *Must always* template and specify the condition as "not (green for priority and green for secondary at the same time)". The condition should look like on the figure below.

After clicking the verify button, UPPAAL returns with the result that our model fails to satisfy the requirement, meaning that there is actually a way to reach the undesired state of letting vehicles come in from every direction. Fortunately, UPPAAL also computes a counterexample (or example in desirable behaviors), demonstrating how exactly we can reach the bad state.



Note: Gamma supports variable model checking algorithms and parameters, that are reachable via the Options menu.



8. Analyzing the Counterexample

The counterexample returned by UPPAAL is not human-readable, and also has a lot of technical details resulting from the generated model, so Gamma will help us decode (*back-annotate*) it. If you check the package explorer again, there is now a *.get* file in folder */trace*. This text file contains a more consumable representation of the counterexample in the following format.

- As the synchronous-reactive semantics is cycle-based, the file contains the states of the specified component (**component** *componentId* after the **import** of its containing package at the beginning of the file) in each particular cycle in block **step**:
    - In block **act**, there are input events that need to be fed to the system ports in order to trigger the behavior. It can be either an
        - event **raise**,

        - time **elapse** (in milliseconds) between the last and the current cycles, or
        - the **schedul**ing of the component.
    - In block **assert**, the state of the component is specified:
        - Lines starting with **raise** list the output events observed on one of the system ports.
        - In block **state** the active state configurations are specified in every component, including complex states.

Note: In block **state** *variable values can also be specified, but the UPPAAL back-annotator does not take variable values into consideration.*

Based on this the problem is related to the *police interrupt* signal and is detailed below.

1. The crossroad initializes and the *crossroad controller* toggles the priority road to enter the *Green* state.
2. A police interrupt arrives, gets relayed to the *traffic light controllers* and switches every light to a blinking yellow state.
3. Two seconds after initializing, the *crossroad controller* gets triggered. After sending out the toggle signals, it now thinks that the priority road has switched to the *Yellow* state, but it is still blinking due to the previous police interrupt.
4. Another one second is elapsed. The *crossroad controller* again sends out the toggle signals and now thinks that the priority road has red, while the secondary road got green light.
5. The policeman changes his mind and sends another interrupt. The signal is relayed to the *traffic light controllers*, returning them into normal operation (priority road green, secondary road red).
6. Another two second later, the *crossroad controller* sends a *toggle* signal to the secondary road believing that it is now turning yellow, but in reality, the *traffic light controllers* are not synchronized anymore, so it gives green light to the secondary road right in the next cycle when it receives the *toggle*.

Based on this scenario, we can see that the problem is in the *crossroad controller*. It should also pause switching states when a police interrupt is in effect.

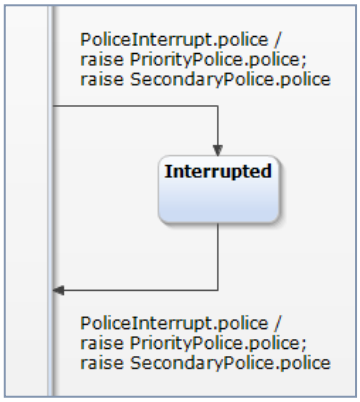
To make sure the counterexample is indeed present in our implementation, Gamma has also generated a test file in */test-gen*. Open it to observe the contents.

In addition to what we have seen before in our hand-made test cases, the generated tests use a *virtual timer service* to measure time. This timer will not use the system time to trigger transitions, it rather simulates time flowing in any rate by calling the *elapse()* method with any amount of time. This is necessary for two reasons. First, using this timer will not pause the test code, so a simulation of one minute will not take longer than a simulation of a millisecond. Secondly, timed counterexamples often represent the corner cases of system behavior, tricky accidents that may happen extremely rarely in real systems. To simulate these cases, we need to have a way for precise simulation of time, e.g., to show that the wrong input in the wrong millisecond can cause great problems.

Run the test cases and make sure they are really present in the implementation. Also notice that we did not have to wait more than a few milliseconds now for the simulation of 5 seconds.

9. Fixing the Controller

As mentioned before, the problem is in the *crossroad controller*. As the last step of this tutorial, let us fix it and re-run the analysis. Extend the */model/Controller/Controller.sct* Yakindu statechart to look like in the figure on the right.



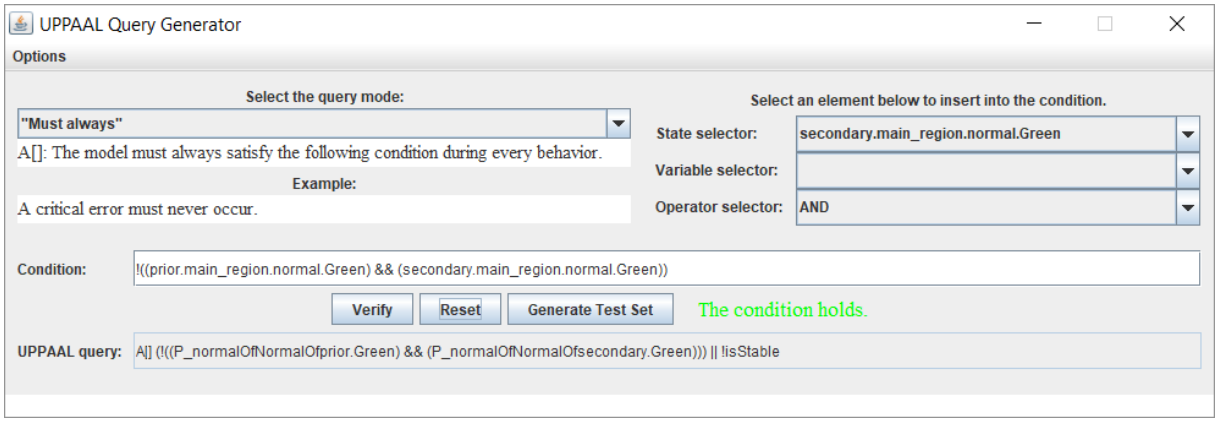
As soon as you save your changes, the Yakindu code generator runs, but we still have to regenerate the Gamma models and the composite implementation, as well as the formal model to re-analyze the system.

Right-click on */models/Controller/Controller.ggen* and select *Gamma Commands > Compile Statechart* to re-compile the statechart.

Right-click on */model/Crossroad.gcd* and select *Gamma Commands > Generate Source Code > Generate Java Code* to re-generate the composite implementation. Re-run the generated test case to make sure the previous counterexample is not executable anymore.

Right-click on */model/Crossroad.gcd* and select *Gamma Commands > Generate Formal Composite Model* to re-generate the UPPAAL model representing the system. Right-click on the file again and open the verification window by selecting *Gamma Commands > Open Query Generator*. Specify the query again to check if there are maybe other ways to reach the undesired state.

If you have done everything correctly, the requirement should now be satisfied, acknowledged by the following view.



10. Exercises

To practice the use of Gamma, we suggest the following exercises.

1. Check the Yakindu statechart of the monitor component modeled in */model/Monitor/Monitor.sct*. The monitor checks the new requirement of not emitting two consecutive *displayRed* or *displayGreen* signals (e.g., because the hardware does not support this). Monitors can be used to intervene in case of erroneous behavior (e.g., shut down the system, or take corrective steps), or to model more complex requirements.
2. Write the Gamma generator model file for the monitor component. Create a new *file* named *Monitor.ggen* and use the Gamma editor to fill it based on the other files present in the project. You may also rely on content assist.
3. Generate the implementation of the Monitor with Yakindu.
4. Using the *.ggen* file, compile the Monitor statechart.
5. Edit */model/Crossroad.gcd* to include the Monitor component in the composite system.
6. Generate code for the composite system with the monitor.
7. Generate the formal model of the composite system with the monitor.
8. Using the model checking functionality, check if the Monitor can every reach the *Error* state during normal operation (assuming there is no hardware fault).
9. *Advanced: Extend the Monitor component with an additional interface. The Monitor should send an error signal when it enters the error state. Do not forget to recompile the interfaces and also the models building on the interface descriptions. Notice the Problems view during the refactoring.*
10. *Advanced: Create a new composite system (MonitoredCrossroad.gcd) consisting of a CrossroadComponent (defined during the tutorial, without the Monitor) and a Monitor component attached to one of the traffic light outputs and its error port published as a system port. Generate the implementation and the formal model, then check if the Monitor can reach the error state, and also if the crossroad controller can reach the SecondaryPrepares state.*