User Guide and Tutorial for the   
**Gamma Statechart Composition Framework**

Adaptive Contracts Extension

# Preliminaries

This document serves as an extension to the original “*User Guide and Tutorial for the Gamma Statechart Composition Framework*” document and reuses its models (*Crossroads* system) to introduce the adaptive contract functionalities of the Gamma framework.

The installation procedure to setup Gamma is the same as specified in the original “*User Guide and Tutorial for the Gamma Statechart Composition Framework*” document. We have prepared a starting project for this tutorial with the name *hu.bme.mit.gamma.tutorial.contract.start*, which should be imported into the Eclipse workspace as an existing project in order to try out the functionalities.

As an extension to the original functionalities, Gamma supports the contract-based testing of reactive systems defined in the Gamma Composition Language (GCL) or Gamma Statechart Language (GSL).

Section 2 presents how simple contracts in the form of scenarios can be specified in the Gamma Scenario Language (GSCL), which is a Live Sequence Chart (LSC) variant with restrictions and extensions to facilitate test generation.

Section 3 introduces how adaptive contracts in the form of statecharts can be specified in the Gamma Adaptive Contract Language (GACL), an extension to the original GSL language. Adaptive contracts specify the activation and deactivation of static scenarios defined in GSCL to support dynamic, context-dependent reconfiguration in adaptive systems.

# Gamma Scenario Language (GSCL)

In a GSCL scenario, the behavior of the component is described based on the observable communication of the component and its environment. The scenario language supports the specification of multiple scenarios in a single file, however, all of them must describe an execution of the same component.

GSCL scenarios consists of *interactions*. In the case of a synchronous component, interactions must be embedded in *synchronous blocks*. These blocks represent execution cycles of the synchronous system. Interactions can be classified based on whether the component *receives* them or *sends* them, whether they are in their *basic form* or *negated*, or by their modality, which can be either *cold* or *hot*.

Concrete executions (interaction sequences) can be classified into *valid*, *inconclusive*, and *invalid* classes based on a specific scenario. If a concrete execution differs from the scenario in a cold interaction, the execution is regarded as *inconclusive* (cold violation), however, if it differs in a hot interaction, the execution is regarded as *invalid* (hot violation). In other words, hot interactions are *compulsory*, while cold interactions are *optional*, and their absence simply implies that the scenario was not specified for the concrete execution. An execution neither valid nor inconclusive is considered *valid*.

Scenarios can also be parametrized and contain references to constants defined in imported *.gcd* files. These values can be used to express concrete values within the scenario.

A cold interaction in a synchronous block received by the component can be specified using the following syntax:

*{cold receives PoliceInterrupt.police}*

Similarly, two interactions sent by the component at the same execution turn is described this way:

*{hot sends PriorityPolice.police   
hot sends SecondaryPolice.police}*

Furthermore, *delays* can be used within scenarios to express the *passage of time* during execution. In the case of synchronous components, delays can be embedded into synchronous blocks or used as a standalone atomic interaction. Delays can be specified using the following syntax where the specified value after the *“delay”* keyword is interpreted as milliseconds:

*{hot delay (500)}*

*or*

*delay (500)*

Scenarios may also contain combined fragments, supporting the compact and high-level specification of the expected behavior. These combined fragments are as follows:

* *optional*: Optional fragments contain a sequence of interactions where the acceptable trace must contain *all of these interactions* or *none* of them. The optional combined fragment can be specified with the following syntax:

*optional {* *{cold receives PoliceInterrupt.police}  
}*

* *alternative*: Alternative fragments contain at least two sequences of interactions. In every execution, *one* of these sets must be part of the concrete execution of the system. Alternative behavior can be specified with the following syntax:

*alternative {  
 {hot sends PriorityControl.toggle}  
} or {  
 {hot sends SecondaryControl.toggle}  
}*

* *loop*: Loop fragments contain a sequence of interactions in addition to a lower and upper limit (specified as integers). An execution trace is acceptable if it contains the content of the loop fragment repeated at least the lower limit times and no more than the upper limit times. The iteration can be described with following syntax:

*loop (1 .. 10) {  
 {cold delay (500 .. 500)}  
 {hot sends PriorityPolice.police   
 hot sends SecondaryPolice.police}  
}*

* *unordered*: Unordered fragments contain at least two sequences of interactions. These sets are handled as atomic and are ordered into every possible permutation: if the concrete execution matches either of these permutations, the trace is accepted. The unordered behavior can be described with the following syntax:

*unordered {*

*{hot sends PriorityControl.toggle}*

*} and {*

*{hot sends SecondaryControl.toggle}*

*}*

* *parallel*: Parallel fragments contain at least two sequences of interactions. The interactions of these steps are ordered into every possible combination, where the only constraint is that the interactions of one sequence should keep their respective order. Similarly, to the unordered combined fragment, if the concrete execution matches either of these combinations, the trace is accepted. The parallel behavior can be described with the following syntax:

*parallel {*

*{hot sends PriorityControl.toggle}*

*} and {*

*{hot sends SecondaryControl.toggle}*

*}*

Moreover, to be able to express, that the component should send specific *events* right after it is started, scenarios can contain an *Initial Outputs Block*, which contains these *events*. The *Initial Outputs Block* can only contain basic *interactions* sent by the component. As all of these *interactions* need to be sent by the component after it is started, the *interactions* are do not need to be embedded in a *synchronous block*. This behavior can be specified with the following syntax:

*initial outputs [*

*hot sends priorityOutput.displayRed*

*hot sends secondaryOutput.displayRed*

*]*

Furthermore, to allow the compact descriptions of complex behavior, the language also supports referencing other scenarios within a scenario. Semantically, referencing a scenario simply means that the body of the referenced scenario is part of the body of the base scenario at the point of the reference. This makes it possible, to define a scenario with a frequently used interaction sequence and simply reference it in other scenarios. For parametrized scenarios, arguments can be passed after the scenarios name between parentheses. Currently recursive referencing of scenarios is not supported. The scenario reference can be used with the following syntax:

*call innerScenario*

## Gamma Scenario Language – Semantic Variations

GSCL supports test generation with configuration options for *1)* specifying constraints for system response and *2)* categorizing unspecified system behavior.

The *AllowedWaiting* annotation describes that the component is not expected to *send* interactions in an exact execution turn, but in an *interval* of execution turns. If the component sends the interaction within the interval, then the concrete execution matches the specified behavior. However, this interval does not apply to interactions *received* by the component since those do not depend on the implementation during testing. The interval can be specified using the following syntax (in the example, the lower limit is zero and the upper limit is one):

*@AllowedWaiting 0 .. 1*

In order to describe the interpretation of *unexpected interactions*, the *Strict* and *Permissive* annotations can be used. In the case of a strict scenario, the concrete execution trace is only acceptable if it contains the exact interactions described in the scenario (interactions in addition to the specified ones are not accepted). However, in the case of permissive scenarios, every trace can be accepted if it contains the described interactions. For a strict behavior use the *@Strict* annotation, while, for a permissive behavior, use the *@Permissive* annotation.

## Test Generation from Static Scenario (GSCL) Contracts

All of the above examples were taken from a scenario called *Example.gsc*, which you can find in the *scenario* folder within the *model* folder of the tutorial project. This folder also contains a scenario called *PoliceInterrupt.gsc*, which describes the response of the *Controller* component in case of a *police interruption*.

Generally, to generate an automaton (Gamma scenario contract statechart) that accepts the *language of the scenario*, you can define a *.ggen* file, which describes the transformation. You can select which scenario you want to formalize, with what name and where the generated statechart should appear. You can also provide values for the scenarios parameters that will be used during the statechart generation.

In the concrete tutorial example, you can find a *StatechartGeneration.ggen* file in the *scenario* folder, with the following content:

*statechart-contract {*

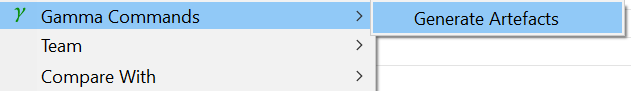
*scenario : PoliceBehaviour*

*folder : "model/scenario"*

*name : "PoliceStatechart"*

*}*

The transformation can be executed by right clicking the *StatechartGeneration.ggen* file and selecting the *Gamma Commands > Generate Artefacts* menu item.



Click on the generated file and select the PlantUML view in Eclipse to visualize the generated scenario contract statechart.

The scenario contract statechart has three special states. The *ColdViolation* state collects *inconclusive* traces, while the *HotViolation* collect *invalid* ones. The *AcceptingState* collects traces that *conform* to the interactions described in the scenarios with respect to the annotations. Every other state represents a point in the scenario, up until which, every interaction of the concrete execution matches the expected interaction. Interactions are represented by complex choice transitions. The figure below shows a model element derived from an interaction sent by the component.

A képen szöveg látható

Automatikusan generált leírás

The transition leading to the choice state fires in every synchronous execution turn. At this state, it is examined whether the expected interactions were sent by the component. If they were, the transition leading to the next state fires. If this transition is unable to fire, the transition leading to the appropriate violation state fires. In the case of a *Strict* annotation, the transition leading to the next state is extended by a guard, which evaluates to true, if no interactions in addition to the specified ones were present.

From these scenario contract statecharts (formalized scenarios), it is possible to generate abstract tests that can be used to verify the component. Generally, this generation also needs to be defined in a *.ggen* file. Within this file, you can set the scenario contract statechart, an output folder relative to the project and an orchestrating constraint if the model has timing (as it does in this example). The transformation can be executed by right-clicking the *.ggen* file, and selecting the *Gamma Commands > Generate Artefacts* menu item.

You can find the *TestGeneration.ggen* file in the folder of the scenarios in addition to the generated statechart. After invoking the artifact generation, a new folder should appear with the name set in the *TestGeneration.ggen* file, containing the generated abstract test. Generally, every test describes an execution trace conforming to the scenario, thus, if any of the generated tests passes, the behavior of the component (with respect to the specified contracts) is accepted.

# Gamma Adaptive Contract Language (GACL)

Gamma also supports the specification of activation and deactivation of scenario contracts during the execution of the system in the context of the Gamma Adaptive Contract Language (GACL). The language is an extension of the GSL language, allowing the specifications of *adaptive contract statecharts*, that is, the activation and deactivation of static GSCL scenarios upon specific events. By building on GSL, GACL supports powerful constructs, such as composite states, parallel regions, history states, variables, and complex transitions, such as choice, fork and join.

As a key feature, GACL supports linking a set of static GSCL *scenarios* to *states*, therefore scenario management can also benefit from the high-level features of statecharts. During the execution of the adaptive contract statechart, the current state configuration indicates the active static scenario set (scenarios linked to the states of the active state configuration in the adaptive contract statechart). As a key semantic characteristic, the GACL statechart has priority over the active static scenario contracts when processing incoming events. When a state configuration is left, the scenarios linked to the left states get deactivated and the ones linked to the newly entered states get activated. As scenarios do not have history, the examination of behavior always starts at the *beginning* of the scenario upon activation.

The following snippet describes an example on how static scenario (GSCL) contracts can be linked to the states of the adaptive statechart:

*…*

*@StatechartContract = Init // Reference to a GSCL contract*

*state Init {*

*entry / set delay := 2 s;*

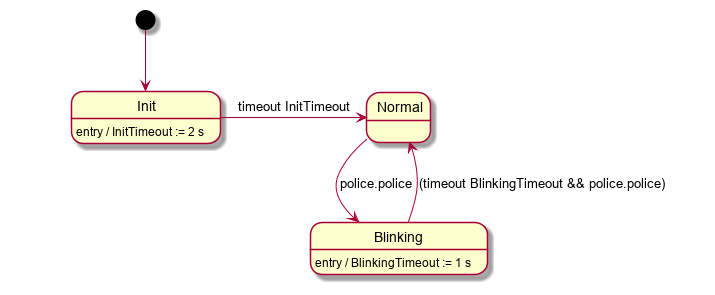
*}*

*…*

## Test Generation from the Adaptive Contract (GACL) Model

You can find an adaptive contract statechart, named *AdaptiveContractCrossroad.gcd* in the *specification* subfolder of the *model* folder in your project.

The figure below shows the *AdaptiveContractCrossroad.gcd* statechart, specifying the behavior of the *Crossroads* system. After start, the system enters its *Initial* phase. Two seconds later, the system starts its normal behavior. In case of a *police.police* event, the behavior changes to *Blinking*, however in case of another *police.police* event and if a second has passed, the behavior changes back to *Normal*.



Furthermore, there is a *Contracts.gsc* file describing three static scenarios specifying the behavior of the *Crossroads* system.

Your task is to complete the scenarios based on the comments and the expected behavior of the system.

From the *Contracts.gsc* file, you can generate Gamma scenario contract statecharts the same way as before. After refreshing the project, the errors in the adaptive statechart should disappear.

Moreover, you can find a file in the same folder, called *Crossroad.ggen*, which describes a configuration for the adaptive test generation based on the specifications. It contains the following specification:

*adaptive-test {*

*analysis {*

*component : AdaptiveContractStatechart*

*language : Theta*

*state-coverage*

*constraint : {*

*minimum-orchestrating-period : 2000 ms*

*maximum-orchestrating-period : 2000 ms*

*}*

*}*

*language : java*

*}*

By right-clicking on the *Crossroad.ggen* file and selecting the *Gamma Commands > Generate Artefacts*, you can generate the abstract tests, which will be output in the *trace* folder of the project. These abstract tests contain the steps of when a scenario should be activated, and the steps generated from the scenario.

From these abstract tests, you can generate concrete tests using the following .*ggen* specification:

*test {*

*trace : CrossroadsTrace*

*language : java*

*}*

Do the generated tests pass? If not, what is the problem? Can you redesign the original statechart models of the system to make the tests pass?