GEMOC Demonstrator - building an executable language for flight controller specification

Table of Contents

Please look at https://gitlab.inria.fr/glose/flight-controller-language-demonstrator for all the sources (including both the FCL language and this document).

The latest online version of this document: as html, as pdf

The goals of this demonstrator is to provide general guidelines about how to use GEMOC technologies in order to build the tooling for a language.

The provided tooling covers several concerns such as:

- Model edition (including textual and graphical edition)
- Model execution (including model debugging).

It is illustrated on a realistic language: a language for designing drone flight controllers.

As explained in https://download.eclipse.org/gemoc/docs/nightly/index.html , building the tooling for a language highlights 2 mains roles: language engineers and modellers.

- Language engineers are the users that design executable DSLs using a metalanguage to define their execution semantics. These users also develop domain- specific tools for their languages, such as editors, validators, compilers… When possible they reuse and/or extend generic tools provided by metalanguage engineers.
- Modelers are the users that design models conforming to executable DSLs. Depending on how supported a given executable DSL and the metalanguage used to define its execution semantics are, modelers will have access to a number of tools to aid them in their endeavor.

The GEMOC Studio used in this demonstrator offers dedicated workspaces and tooling for each of these roles that are referenced as **GEMOC Language workbench** for the tools for *language engineers and **GEMOC Modeling Workbench** for the tools for _Modelers*.

In this document, we may have to switch between these workbench in order to show the result of the language workbench in the final modeling workbench:

- items or roles relative to the Language Workbench or involving the Language Engineer are identified with the following icon :
- items or roles relative to the Modeling Workbench or involving the Modeler are identified with the following icon :

This switch is required only during the development, once finished, one can build a standalone eclipse product dedicated to the Modelers that contains only the final dsl tooling.

For FCL, one can use the following update site to add FCL tooling to an existing Eclipse and get a Modeling workbench.

- https://glose.gitlabpages.inria.fr/flight-controller-language-demonstrator/updatesite/latest/

This document presents the actions in a sequential order corresponding roughly to the first time each of these actions was started. However, the development is actually iterative. Actions of the last steps may require changing or improving element done in previous steps.

Chapter 1. Flight Controller Language - FCL

1.1. Language Overview

When building a flight controller, most people directly write the controller logic in the target platform language. For example, they write in c when targeting ardiuno hardware.

Thanks to GEMOC we build a language dedicated to specify the logic for such code.

A end user of such language would then have a higher level of abstraction and better editing support allowing her to be more efficient when writing complex logic.

The FCL is intended to specify relatively small actions (which might be complex in terms of control command laws on the rotors and sensors) and basic plans. These actions will be organized in modes., and how to switch from one mode to another, and which concrete actions are run. For instance, typical basic modes that would be defined for a copter would be inspired by the ones defined in ardupilot flight controller http://ardupilot.org/copter/docs/flight-modes.html.

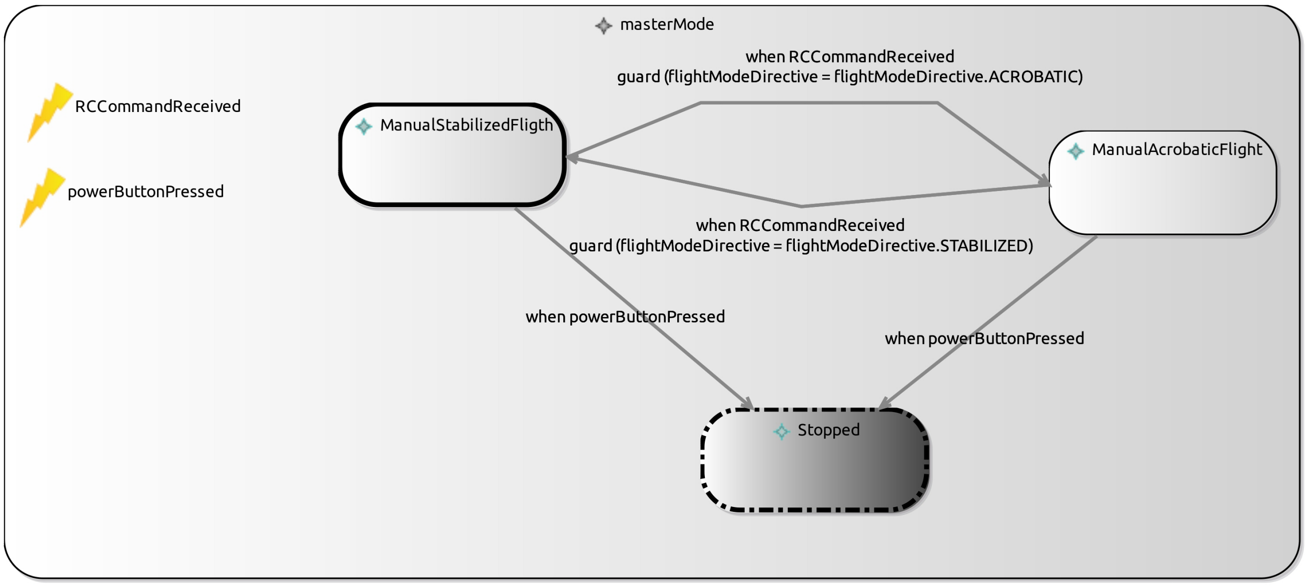


Figure 1. Mini quadcopter controller mode diagram

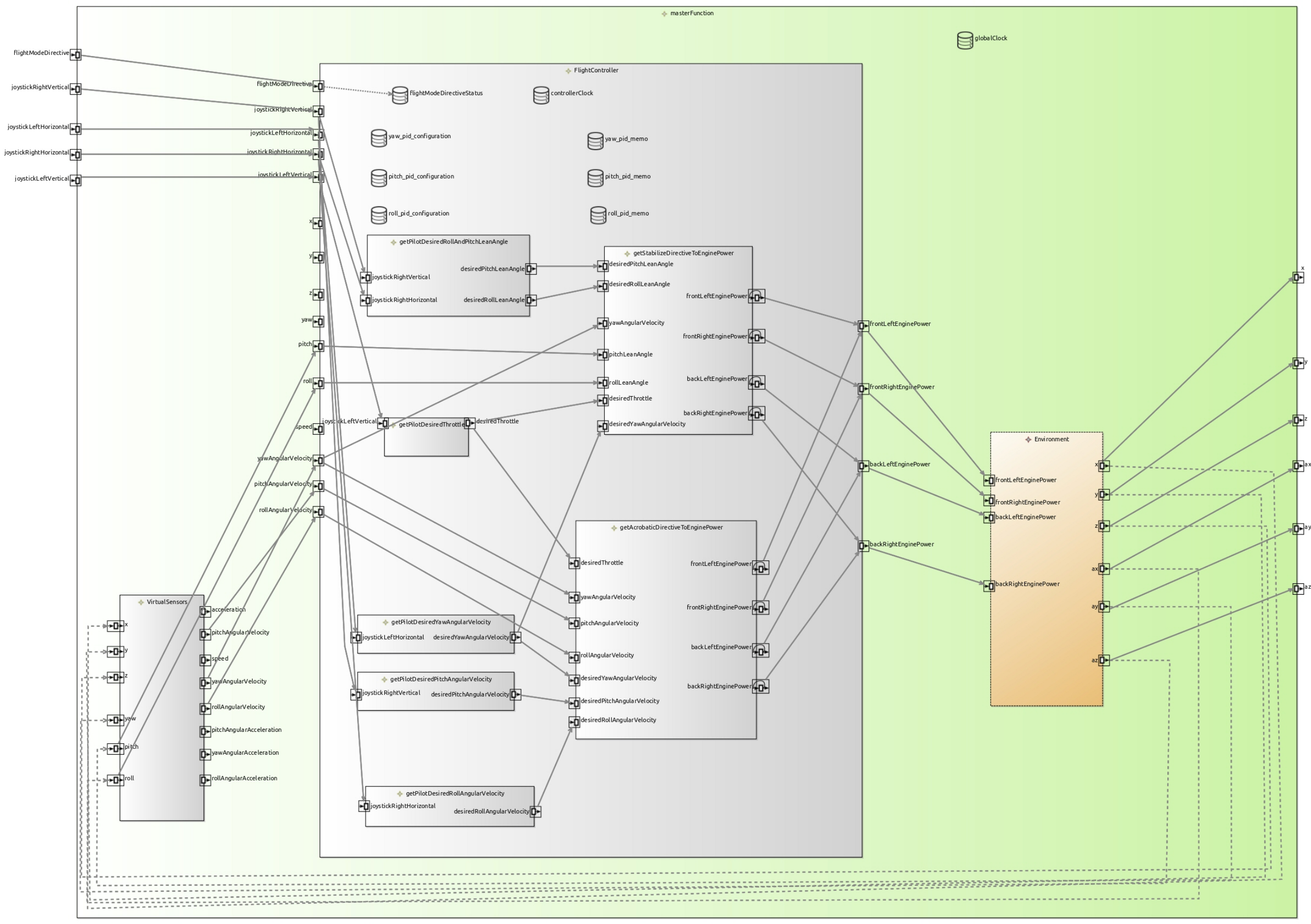


Figure 2. Mini quadcopter controller function diagram

1.2. Repository Content Summary

The sources in git contains the following content:

1.2.1. FCL Specifications

The specifications of FCL are in fcl-implementations.

The folder `fcl` contains the projects building a language that is not executable yet, (ie. the common part of various behavior implementation)

- fcl/fr.inria.glose.fcl.dsml project contains the GEMOC declaration of the fcl common part. It is mainly used to drive the helper wizards;
- fcl/fr.inria.glose.fcl.model project contains the structural definition of the language (see Section 2.1);
- fcl/fr.inria.glose.fcl.design contains the graphical editor specification of the language (see Section 2.2.2);
- fcl/fr.inria.glose.fcl.xtext contains textual editor specification of the language (see Section 2.2.1);
- fcl/fr.inria.glose.fcl.xtext.ide and fcl/fr.inria.glose.fcl.xtext.ui contain the code for integrating the textual editor in Eclipse;
- fcl/fr.inria.glose.fcl.xtext.tests and fcl/fr.inria.glose.fcl.xtext.ui.tests projects contains a base for unit tests.

The folder `commons` contains the projects that are not specific to FCL

- commons/org.javafmi project contains the java driver allowing to load FMI/FMU;

The folder `alefcl` contains the projects defining an executable language for FCL writing using ALE.

- alefcl/fr.inria.glose.fcl.alefcl.xdmsl project contains: the GEMOC declaration of the executable FCL: ALEFCL; the ALE extension that adds the behavioral semantics on top of the FCL base ().
- alefcl/fr.inria.glose.fcl.alefcl.xdmsl.design project contains the debug and animation extension to fr.inria.glose.fcl.design dedicated to the ALEFCL language.
- alefcl/fr.inria.glose.fcl.alefcl.vm.mdodel project contains some additional extension written in ecore defining some VirtualMachine concepts used by ALEFCL.

1.2.2. Demonstration Controllers

Some FCL model examples are located in: model_examples.

The demonstrator main example is located in fr.inria.glose.miniquadcoptercontroller.demo_v0. It models a Quadcopter flight controller with two modes: a manual acrobatic mode and a manual stabilized mode.

Chapter 2. Scenario #1: Building an Executable Modeling Language

This scenario demonstrates the ability of the GEMOC Studio to help a language designer to build her own modeling language, including the concepts (aka. abstract syntax), the textual and/or graphical representation (aka. concrete syntax), and the meaning of those concepts over the time (aka. behavioral semantics). The behavioral semantics is specified in the form of a virtual machine allowing the execution of a conforming model.

As an outcome of such a scenario, the demonstrator shows how to automatically get a domain-specific modeling and execution environment from the specification of the modeling language. The use of resulting environment is demonstrated in Chapter 3

2.1. FCL Metamodel Design

During this phase the Language engineer captures the concepts the Modeler will be able use.

The main concerns of the Language engineer during this step will be to specify a class diagram with a special focus on:

- the list of concepts
- their relationship. This includes the containment relationship which is used by many of the generic tools in order to provide efficient default implementations.

2.1.1. FCL Ecore

⇒

Basically, FCL allows to model a DataFlow where the enabled Functions are filtered by a StateMachine defining "Modes".

Figure 3 shows the relations between the major concepts:

- `FCMModel`, the root of the model;
- `ModeStateMachine`, the state machine controlling modes;
- `Function`, the functions of the dataflow;
- `FunctionPort`, the input and ouput port of functions;
- `FunctionConnector`, the connections between ports;
- `Event`, the event that can be used to trigger mode change;
- `DataType`, the primitive type of the data used in the model;
- `Action`, the action done when processing functions or when triggering mode changes;
- `Expression`, the expression used in actions and in the statemachine guards.

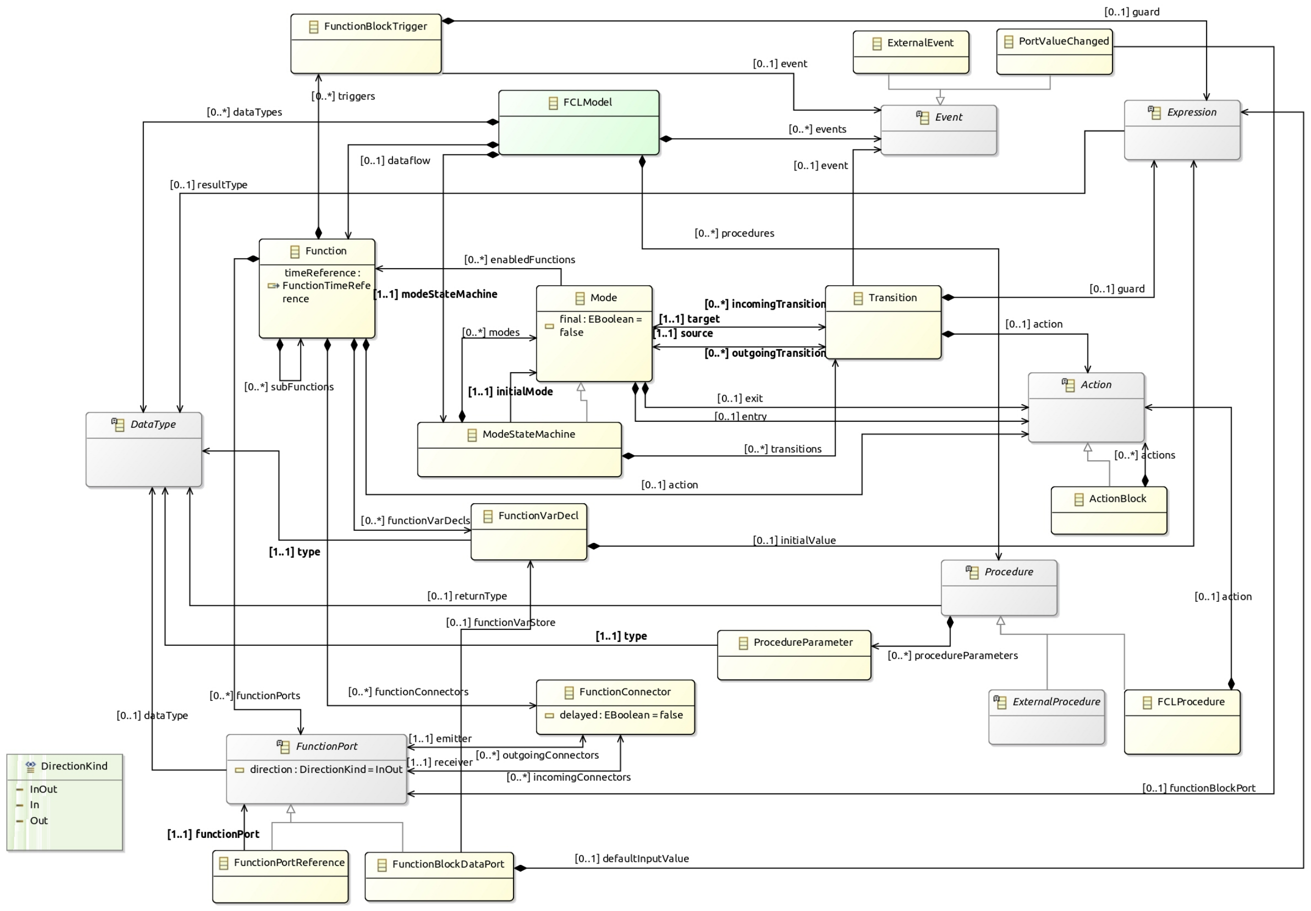


Figure 3. FCL main class diagram

Figure 4 shows the *ModeStateMachine* concepts. Basically, it models a statemachine controlling enabled Functions. The *ModeStateMachine* is composed of *Modes* that enable a set of *Function*. Changing mode is done by going through *Transition* where these *Transition* are guarded by an *Event* and/or an *Expression*. Some *Action* can be defined when a *Transition* is fired or when entering or leaving a *Mode*.

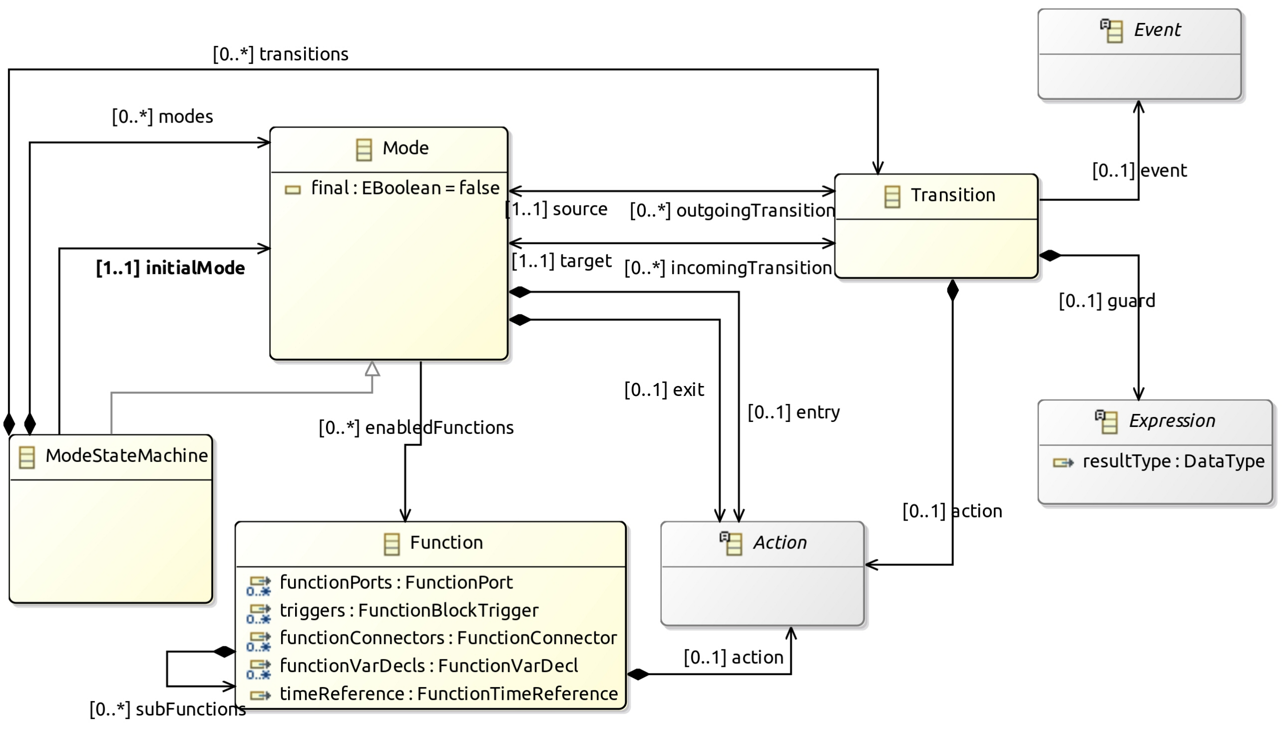


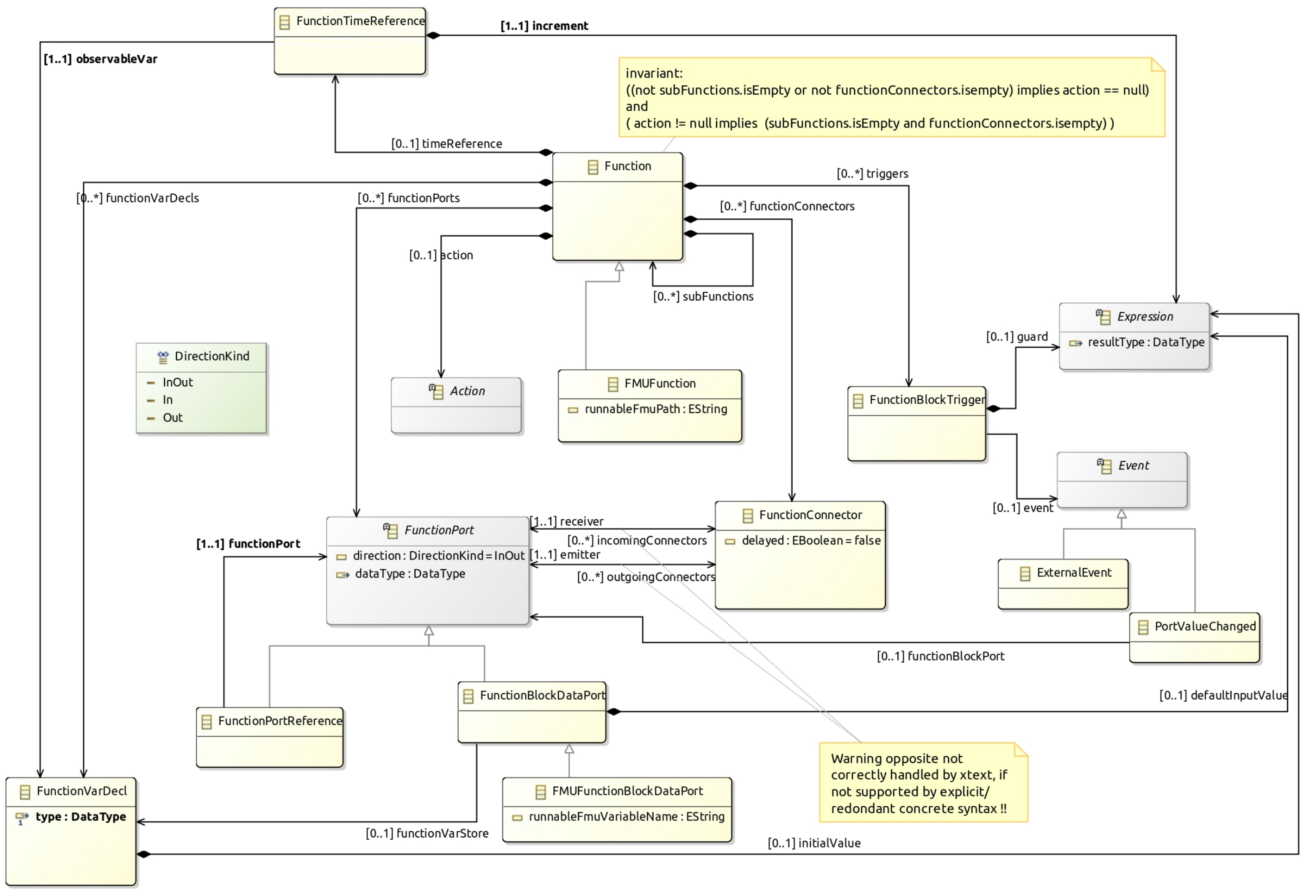
Figure 4. StateMachine class diagram

Figure 5 shows the *Function* concepts. Basically, it models a data flow between *Functions*. These *Functions* declare *DataPorts* that can be connected through *FunctionConnectors*. Each Port may be In, Out or InOut. A *Function* can declare sub functions.

FMUFunction and *FMUFunctionBlockDataPort* are specializations dedicated to handle the connect a blackbox *Function* that can communicate using FMI/FMU protocol.

Basic *Function* declare an *Action* whose role is to specify the value to set on Out ports from In ports.

Functions can declare variables that are local to the function. It allows to use them (read/write) from *Expressions* and *Action* (ie. Event, procedure body, …). They can be associated to a port but can also have their own modification lifecycle (for example, a nested function modifying a value in its parent function…).



invariant:
((not subFunctions.isEmpty or not functionConnectors.isempty) implies action == null)
and
(action != null implies (subFunctions.isEmpty and functionConnectors.isempty))

Warning opposite not correctly handled by xtext, if not supported by explicit/ redondant concrete syntax !!

Figure 5. BlockFlow class diagram

Figure 6 shows the concepts supported to define the *Expression* in various places in the DSL. *Expression* are typed, this allows several type verification either statically and at runtime (see also Figure 8).

BinaryExpression defines a set of operators allowing to combine 2 expressions.

UnaryExpression defines some simple operation on a given expression.

CallConstant allows to define constant in the model (for each of the supported data type).

CallDeclarationReference allows to get the current value associated to one of the *CallableDeclaration* elements of the language: a port of a *Function* (*FunctionPort*), a local variable declaration (*VarDecl*), a variable in a function (*FunctionVarDecl*) or an enumration literal.

Similarly to *CallDeclarationReference*, *CallPrevious* allows to get the previous value associated to one of the *CallableDeclaration* elements of the language.

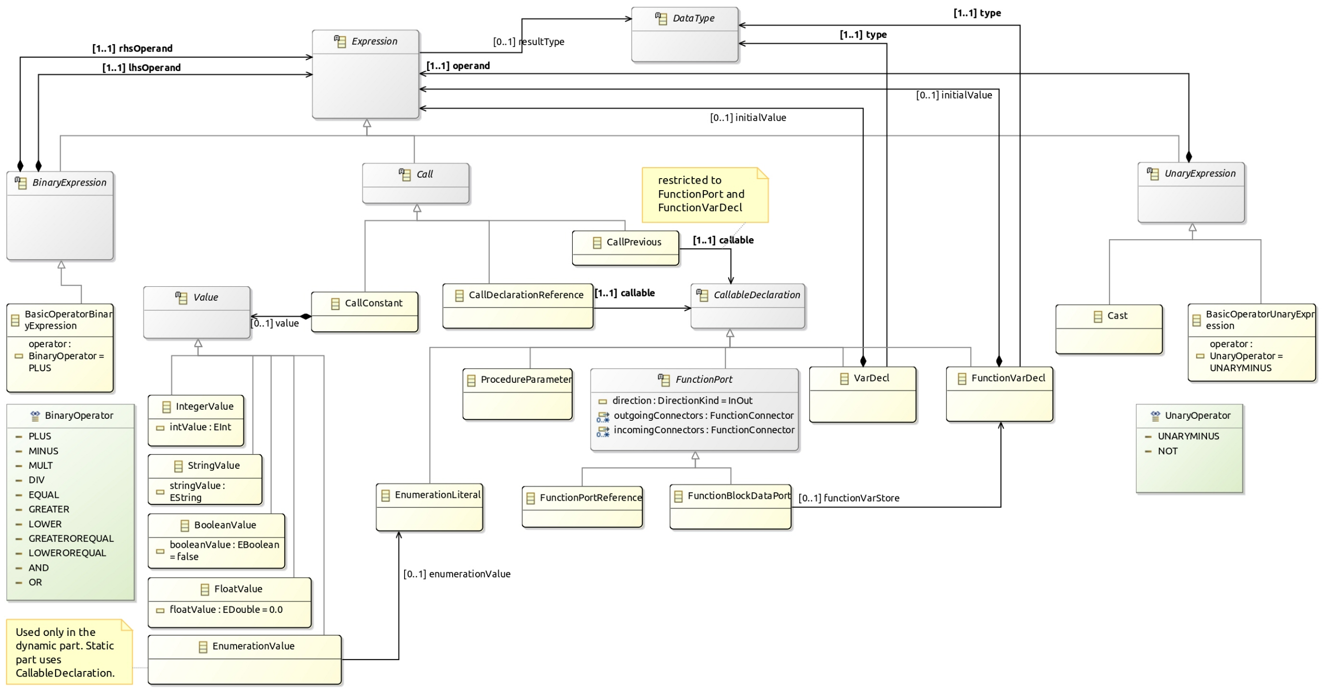


Figure 6. Expression class diagram

Figure 7 shows the concepts supported to define the *Action* in various places in the DSL. It basically uses *Expression* in order to provide the value to set in function ports and function variables.

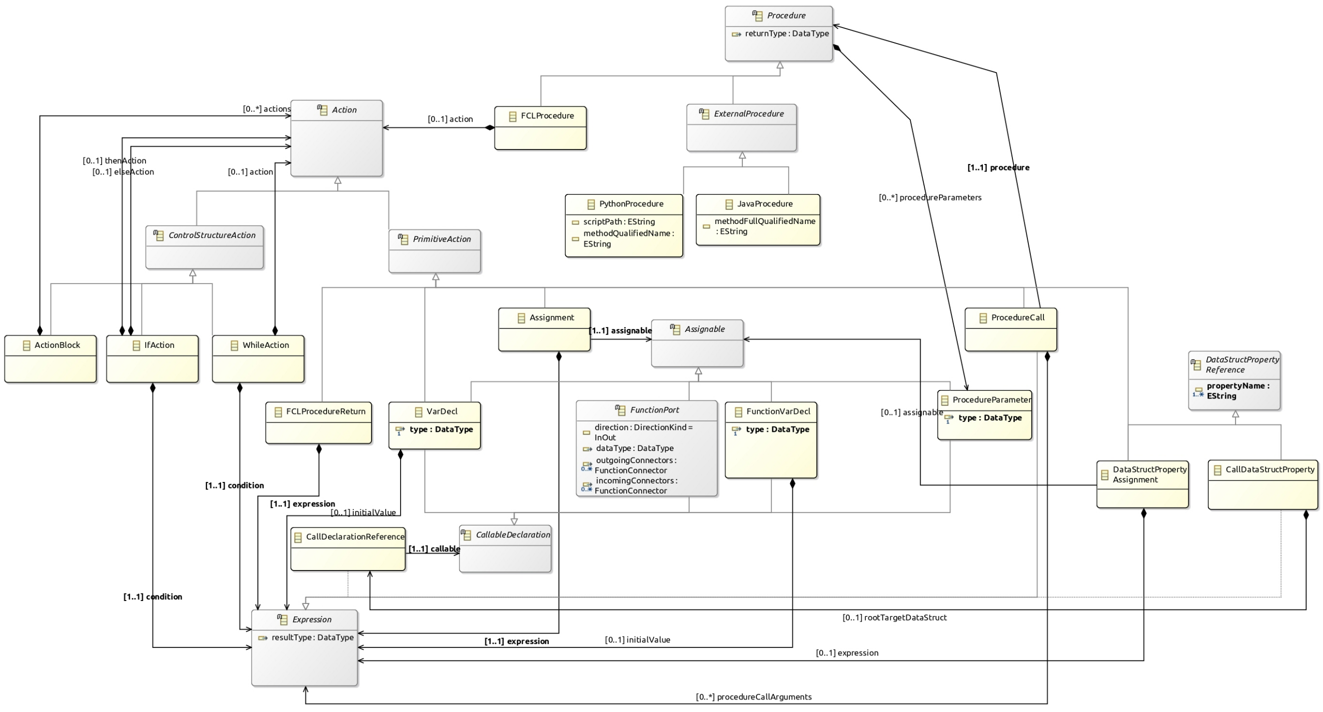


Figure 7. Action class diagram

Figure 8 shows the primitive types that the Modeler can use when she define types in the model. These types allow validation and early error reporting when associated with validation rules (see Section 2.2.4).

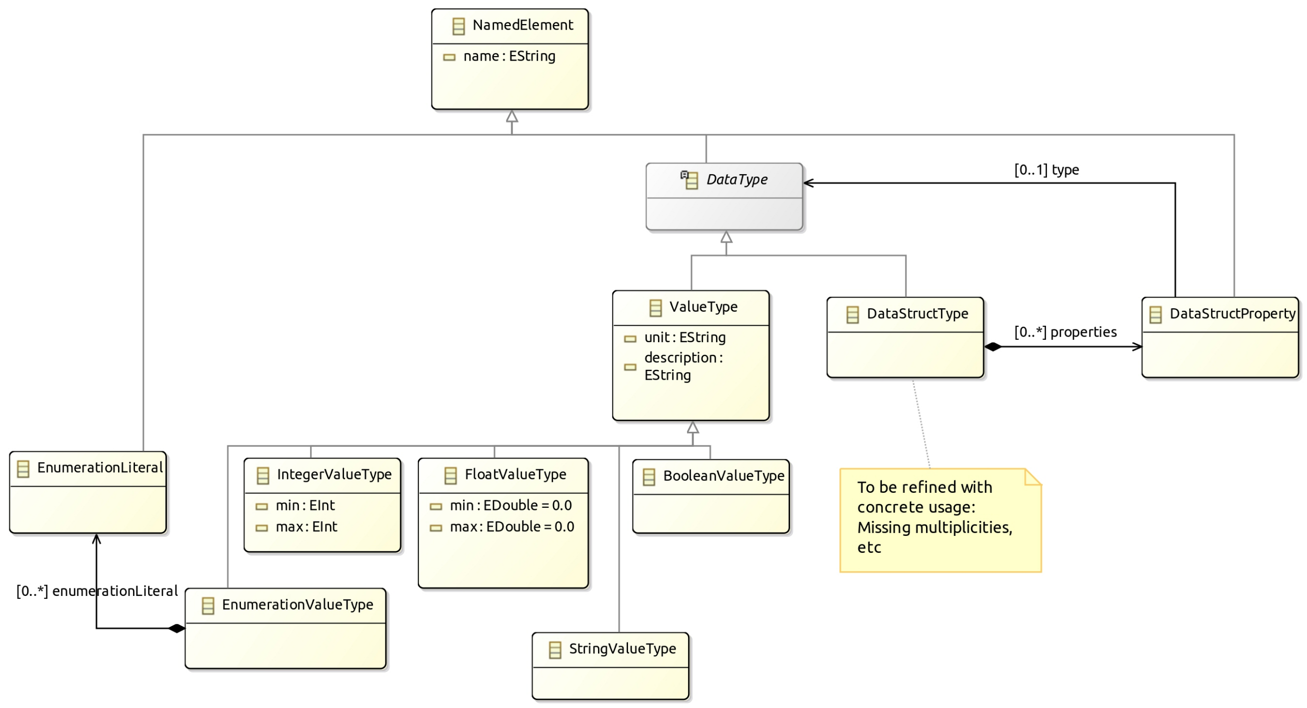
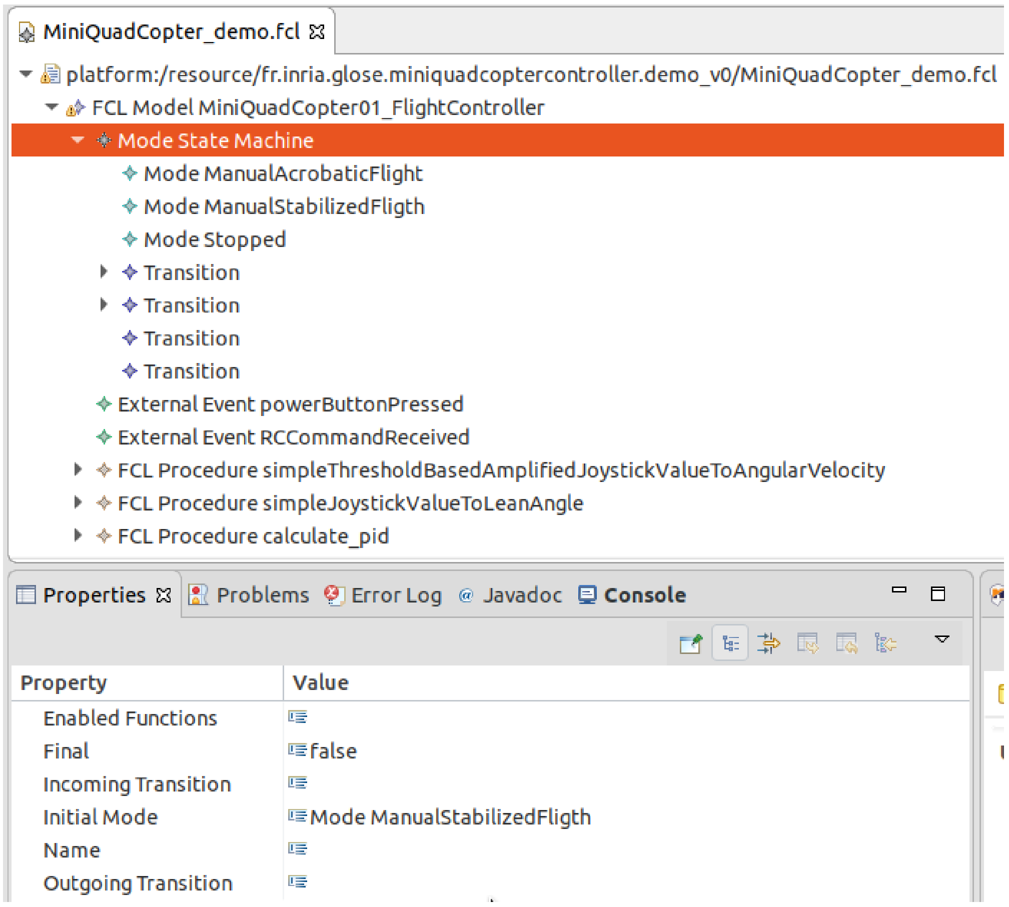


Figure 8. Types class diagram

2.1.2. Basic Tree Editor

⇒

Up to this point, the ecore provide a minimalistic model edition support using a tree representation. Figure 9 shows this tree editor on the basic quadcopter controller.


MiniQuadCopter_demo.fcl

platform:/resource/fr.inria.glose.miniquadcoptercontroller.demo_v0/MiniQuadCopter_demo.fcl
- FCL Model MiniQuadCopter01_FlightController
  - Mode State Machine
    - Mode ManualAcrobaticFlight
    - Mode ManualStabilizedFligth
    - Mode Stopped
    - Transition
    - Transition
    - Transition
    - Transition
  - External Event powerButtonPressed
  - External Event RCCommandReceived
  - FCL Procedure simpleThresholdBasedAmplifiedJoystickValueToAngularVelocity
  - FCL Procedure simpleJoystickValueToLeanAngle
  - FCL Procedure calculate_pid

Property	Value
Enabled Functions	
Final	false
Incoming Transition	
Initial Mode	Mode ManualStabilizedFligth
Name	
Outgoing Transition	

Figure 9. DemoController model in tree editor screenshot

This tree editor is convenient during the early steps of the ecore definition as is allows to create sample models and manually check the expressivity of the language.

2.2. Editors Design

The demonstrator illutrates the developement of two editors that collaborates:

- a textual editor allowing to fully write the model in a modern text editor with completion, syntax highlighting, error/warning reporting.
- a graphical editor allowing to display and edit some part of the model. This editor doesn't intend to provide a graphical representation for all concepts but will open the textual editor when required. (for example, the `Expression` or `Action` part of the model are displayed/edited using a textual syntax only)

Even if this demonstrator focuses on more sophisticated editors (Textual and graphical),

it is easy to customize the basic tree editor (see Section 2.1.2) to provide nicer labels and icons.

Simply do a *generate Model edit* and *generate Model editor* on the genmodel file. Then change the gif images and edit the `getImage` and `getLabel` methods in the `*ItemProvider` java classes.

Do not forget to add a `Generated NOT` in the comment of the modified methods to prevent overriding of your code if you regenerate the code due to an update in the metamodel.

2.2.1. XText Editor

Starting an XText editor for FCL is done by doing:

- Right Click on the dsl project
- *GEMOC Language → Create Xtext editor project for Language*
  - project name: fr.inria.glose.fcl.xtext
  - Name: fr.inria.glose.fcl.xtext.FCL

You can also do a *File → new → project → xtext project from existing ecore models*

This creates the project: `fr.inria.glose.fcl.xtext`

The file `fr.inria.glose.fcl.xtext/src/fr/inria/glose/fcl/FCL.xtext` contains the grammar allowing to parse *.fcl files.

The default grammar rules generated by the project creation wizard are functional but doesn't suit our needs. So they are customized to provide a friendlier textual syntax.

A Right click on the xtext file → *Run As → Generate Xtext artifacts* will update the java code for the parser.

reference documentation https://www.eclipse.org/Xtext/documentation/301_grammarlanguage.html

After modification (see `fr.inria.glose.fcl.xtext/src/fr/inria/glose/fcl/FCL.xtext`), it now supports operator precedence, and a syntax less verbose than the default one.

Xtext allows to define formatting rules. These rules readjusts the spaces, indentation and line breaks when using `Ctrl`+`Shift`+`F` in the editor. The formatter code in conveniently written in the form of rules in FCLFormatter.xtend

While providing formatting rules is only a minor addition when using only xtext editor, this becomes important to have it if the graphical views also allow to edit the model. This ensures that the newly created elements will be properly displayed in the textual editor.

⇒ Section 3.1.1 shows the resulting XText editor in action.

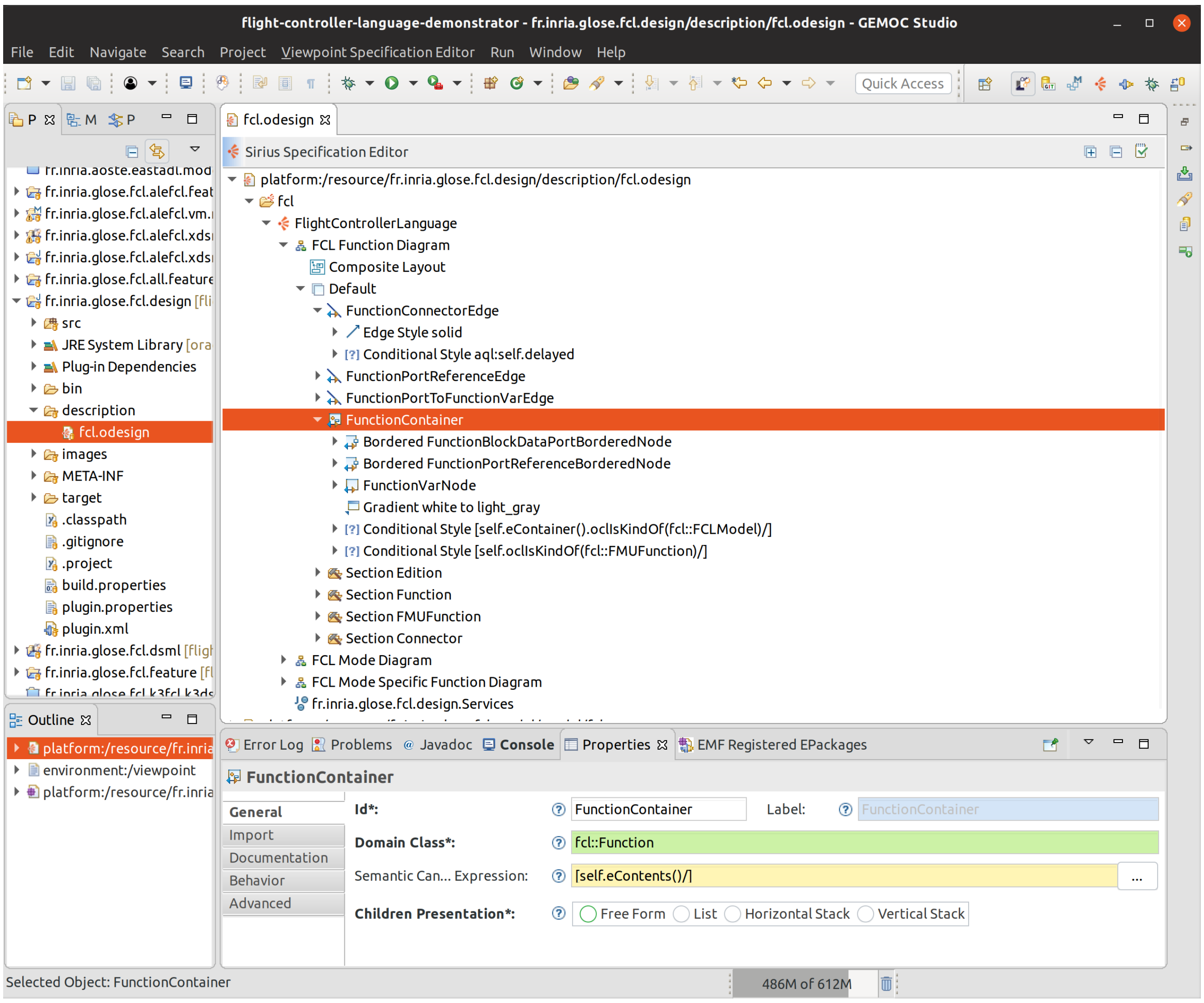
2.2.2. Sirius Editor

To initiate a first diagram for FCL:

- Right Click on the dsl project
- *GEMOC Language* → *Create Sirius editor project for Language*
    - project name: fr.inria.glose.fcl.design
- *Next* → *Next* → check the box *create a plugin using one of the following template* → Select FCLModel as root container → *Finish*

The Sirius diagram specification is in fr.inria.glose.fcl.design project.

The .design of the demonstrator defines 2 diagrams: * one diagram dedicated to show the StateMachine of the Mode * one diagram dedicated to show the DataFlow between the Function


flight-controller-language-demonstrator - fr.inria.glose.fcl.design/description/fcl.odesign - GEMOC Studio
File Edit Navigate Search Project Viewpoint Specification Editor Run Window Help
Quick Access
fcl.odesign
Sirius Specification Editor
Outline
Error Log Problems Javadoc Console Properties EMF Registered EPackages
General
Import
Documentation
Behavior
Advanced

```
fr.inria.aoste.eastadl.mod
fr.inria.glose.fcl.alefcl.feat
fr.inria.glose.fcl.alefcl.vm.
fr.inria.glose.fcl.alefcl.xds
fr.inria.glose.fcl.alefcl.xds
fr.inria.glose.fcl.all.feature
fr.inria.glose.fcl.design [fli
  src
  JRE System Library [ora
  Plug-in Dependencies
  bin
  description
    fcl.odesign
  images
  META-INF
  target
  .classpath
  .gitignore
  .project
  build.properties
  plugin.properties
  plugin.xml
fr.inria.glose.fcl.dsml [fligh
fr.inria.glose.fcl.feature [fl
fr.inria.glose.fcl.k3fcl.k3ds
```

```
platform:/resource/fr.inria.glose.fcl.design/description/fcl.odesign
  fcl
    FlightControllerLanguage
      FCL Function Diagram
        Composite Layout
        Default
          FunctionConnectorEdge
            Edge Style solid
            [?] Conditional Style aql:self.delayed
          FunctionPortReferenceEdge
          FunctionPortToFunctionVarEdge
          FunctionContainer
            Bordered FunctionBlockDataPortBorderedNode
            Bordered FunctionPortReferenceBorderedNode
            FunctionVarNode
            Gradient white to light_gray
            [?] Conditional Style [self.eContainer().oclIsKindOf(fcl::FCLModel)/]
            [?] Conditional Style [self.oclIsKindOf(fcl::FMUFunction)/]
          Section Edition
          Section Function
          Section FMUFunction
          Section Connector
      FCL Mode Diagram
      FCL Mode Specific Function Diagram
      fr.inria.glose.fcl.design.Services
```

```
platform:/resource/fr.inria
environment:/viewpoint
platform:/resource/fr.inria
```

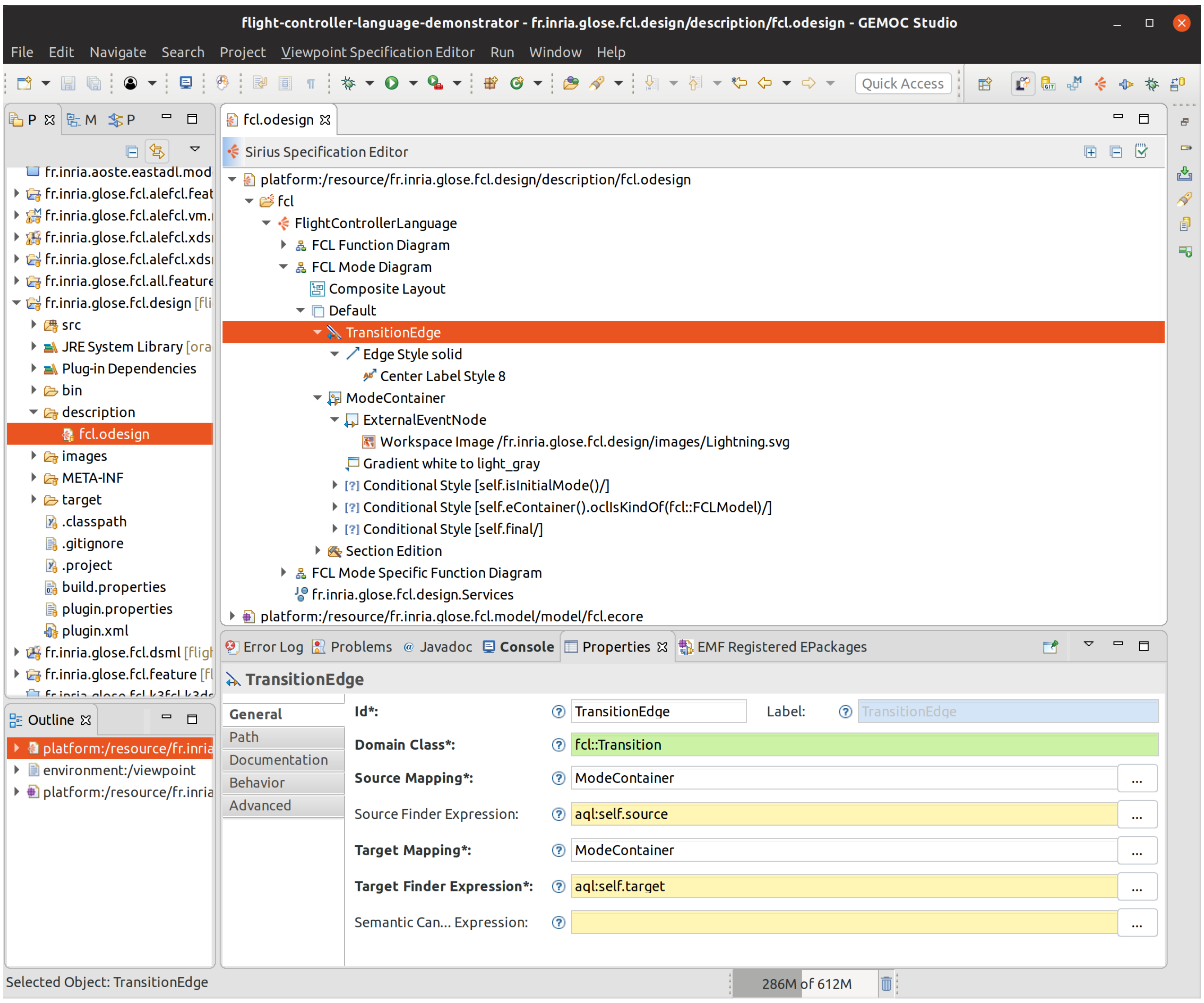
FunctionContainer

Id*:	FunctionContainer	Label:	FunctionContainer
Domain Class*:	fcl::Function		
Semantic Can... Expression:	[self.eContents()/]		
Children Presentation*:	Free Form List Horizontal Stack Vertical Stack		

Selected Object: FunctionContainer

486M of 612M

Figure 10. FCL Mode diagram specification screenshot



flight-controller-language-demonstrator - fr.inria.glose.fcl.design/description/fcl.odesign - GEMOC Studio

File Edit Navigate Search Project Viewpoint Specification Editor Run Window Help

```
fr.inria.aoste.eastadl.mod
fr.inria.glose.fcl.alefcl.feat
fr.inria.glose.fcl.alefcl.vm.
fr.inria.glose.fcl.alefcl.xds
fr.inria.glose.fcl.alefcl.xds
fr.inria.glose.fcl.all.feature
fr.inria.glose.fcl.design [fli
  src
  JRE System Library [ora
  Plug-in Dependencies
  bin
  description
    fcl.odesign
  images
  META-INF
  target
  .classpath
  .gitignore
  .project
  build.properties
  plugin.properties
  plugin.xml
fr.inria.glose.fcl.dsml [fligh
fr.inria.glose.fcl.feature [fl
```

Outline

```
platform:/resource/fr.inria
environment:/viewpoint
platform:/resource/fr.inria
```

fcl.odesign — Sirius Specification Editor

```
platform:/resource/fr.inria.glose.fcl.design/description/fcl.odesign
  fcl
    FlightControllerLanguage
      FCL Function Diagram
      FCL Mode Diagram
        Composite Layout
        Default
          TransitionEdge
            Edge Style solid
              Center Label Style 8
          ModeContainer
            ExternalEventNode
              Workspace Image /fr.inria.glose.fcl.design/images/Lightning.svg
              Gradient white to light_gray
              [?] Conditional Style [self.isInitialMode()/]
              [?] Conditional Style [self.eContainer().oclIsKindOf(fcl::FCLModel)/]
              [?] Conditional Style [self.final/]
          Section Edition
      FCL Mode Specific Function Diagram
      fr.inria.glose.fcl.design.Services
platform:/resource/fr.inria.glose.fcl.model/model/fcl.ecore
```

Error Log | Problems | Javadoc | Console | Properties | EMF Registered EPackages

TransitionEdge

General | Path | Documentation | Behavior | Advanced

Id*:	TransitionEdge
Label:	TransitionEdge
Domain Class*:	fcl::Transition
Source Mapping*:	ModeContainer
Source Finder Expression:	aql:self.source
Target Mapping*:	ModeContainer
Target Finder Expression*:	aql:self.target
Semantic Can... Expression:	

Selected Object: TransitionEdge   286M of 612M

Figure 11. FCL DataFlow diagram specification screenshot

TODO add explanation

Several of the expressions in the odesign are simplified by using a java service. The services are defined in fr.inria.glose.fcl.design/src/fr/inria/glose/fcl/design/Services.java.

Reference documentation for writing queries in Sirius: https://www.eclipse.org/sirius/doc/specifier/general/Writing_Queries.html https://www.eclipse.org/acceleo/documentation/aql.html https://www.eclipse.org/acceleo/documentation/

⇒ Section 3.1.2 shows the resulting Sirius editor in action.

Similarly to programming where several programs may produce the same behavior, it exist several ways to write Sirius diagram specification that draw the same diagram.

In this demonstrator some attempt have been done to provide correct reuse of the rules and keep the global design simple. However, it can probably be improved. If you see improvement that do not blur the teaching goal feel free to contact the author and propose a change ☺.

2.2.3. Sirius/Xtext Integration

TODO write section

The java service fr.inria.glose.fcl.design/src/fr/inria/glose/fcl/design/Services.java defines several methods that helps Sirius/XText collaboration. For example, the `xtextPrettyPrint()` (see Listing xtextPrettyPrint java service) that allows to get the XText representation of an EObject (including its contained element) is very handy to represent Expression in labels.

Listing xtextPrettyPrint java service

```
public String xtextPrettyPrint(EObject any) {
    if (any != null && any.eResource() instanceof XtextResource &&
any.eResource().getURI() != null) {
        String fileURI = any.eResource().getURI().toPlatformString(true);
        IFile workspaceFile = ResourcesPlugin.getWorkspace().getRoot().getFile(new
Path(fileURI));
        if (workspaceFile != null) {
            ICompositeNode node = NodeModelUtils.findActualNodeFor(any);
            if (node != null) {
                return node.getText().trim();
            }
        }
    }
    return "";
}
```

The method `openBasicHoveringDialog(EObject any)` is convenient to open a popup window for long text that doesn't fit in a label.

The method `openTextEditor(EObject any)` that allows to open the XText editor is useful to open the selected element in the text editor. As the internal EMF resources are connected, saving changes done in the text editor are automatically taken into account in the graphical editor.

XText/Sirius collaboration work correctly only if the user save doesn't try to do modification both in the textual editor and the graphical editor at the same time. She must save the model before trying to edit using another editor.

2.2.4. Static Validation

The structure of the metamodel defined in Section 2.1 already provides some guidance to the Modeler in order to build correct models. However, this structure cannot capture all non valid situations. Many of them can be checked early by writing validators.

Several technique are possibles (OCL, Sirius validation rule, Xtext validation framework, EMF validation framework).

In this demonstrator, as it relies on Xtext for the serialization, we've implemented a set of check rules using Xtext framework which is very easy to set up. They are in the following file fr.inria.glose.fcl.xtext/src/fr/inria/glose/fcl/validation/FCLValidator.xtend.

They cover:

- type checking of *Expression* and *Action*
- checking of call/assignment of properties in complex data types
- checking of number of parameter when calling a *Procedure*
- checking of initial value on variables
- checking of loop connection (ie. InOut ports that have a connection to themselve which are valid only if the connector is delayed, that it, its value is transmitted only when finishing the parent function)

Many other useful check rule could be written, the Language engineer can write rule detecting recurrent errors that many Modelers do.

In some case, the check might be only some advice reported as warning. For example, in FCL, a rule report a warning when 2 different types are compatibles but combined (or assigned) together. This helps the Modeler in focusing on places with potential conversion errors.

2.3. Behavioral Semantic and Debugger Design

TODO Write section explain ALE, explain FCL behavior

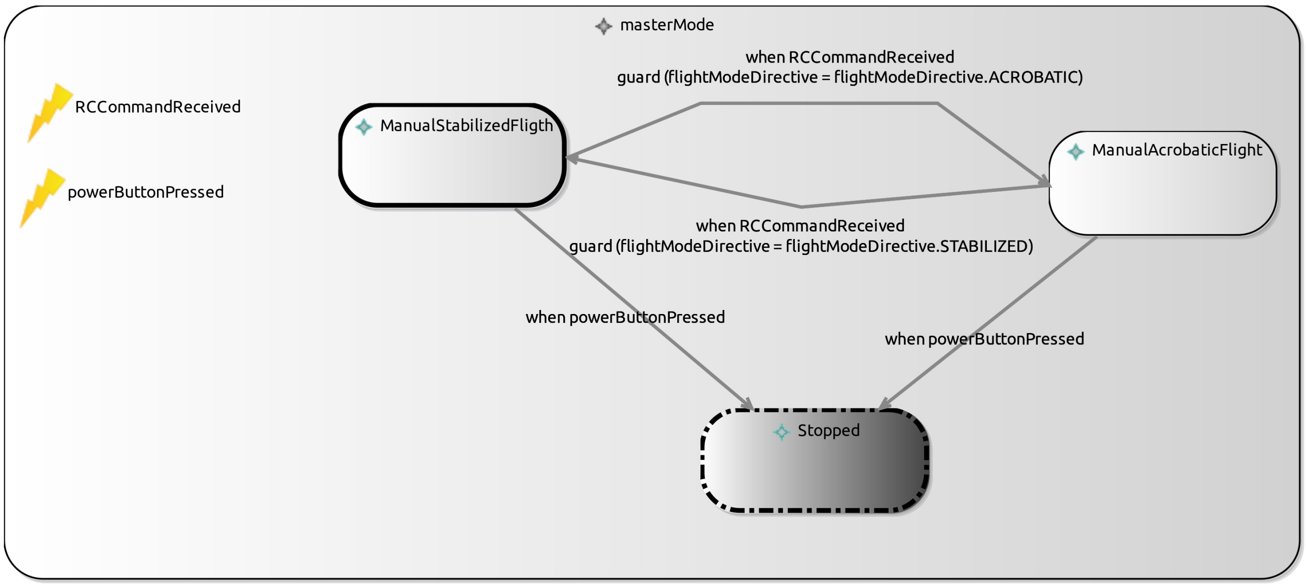


Figure 12. Mini quadcopter controller mode diagram

2.4. FCL Animation

The Sirius diagram animation specification is in fr.inria.glose.fcl.alefcl.xdsml.design project.

It extends the fr.inria.glose.fcl.design project (see Section 2.2.2) by adding new layers.

A first layer **Debug** has been added to support interaction with the GEMOC debugger. It role is to provide:

- breakpoint representation and actions (toggle breakpoint),
- engine interaction, allowing to start an execution
- execution stack highlighting (ie. when clicking a step in the execution stack, it highlights the step target model elements)

A second layer **Animation** has been added to support domain specific (FCL) animation directly on top of the model diagrams.

> As the graphical representation for FCL is composed of 2 diagrams (Mode and DataFlow), the new layers **Debug** and **Animation** must be specified in each of these diagrams.

2.5. Extra views

The demonstrator also provide a FCL specific view: FCL External Ports View.

This view is built as an eclipse UI view that registers to the execution as a GEMOC engine addon dedicated only to FCL.

This view is designed to show only the outer FunctionPorts of the running model.

It also allows to modify the values of these FunctionPorts.

The java code of this view is in the fr.inria.glose.fcl.alefcl.xdsml project.

- ExternalPortsView.java contains the code for the UI
- ExternalPortsViewAddon.java is the addon that listen for GEMOC engine event and trigger an update of the view.
- plugin.xml registers the view to Eclipse UI and registers the addon to the GEMOC Engine specifically for FCL language.

2.6. Deployment

The modeling environment can be deployed using several way:

- for development purpose, a runtime workbench is started from the language workbench. It allows to test and debug the language tooling.
- for use by the final end user (ie. the Modeler), the projects can be packaged and installed directly in an Eclipse product.

2.6.1. Packaging and Deployment in an Eclipse Product

TODO write section

2.6.2. Runtime Deployment

To get a runtime deployment for testing a language:

- *Run* → *Run configurations…* → Right click on Eclipse Application → *new configutation*

This creates a launch configuration that runs a new Eclipse similar to the current one but also that also includes the plugins under development in the workspace.

Chapter 3. Scenario #2: Simulating and Debugging a Behavioral Model

This scenario demonstrates the ability of the GEMOC Studio to provide a full-fledged modeling and execution environment, that supports a language user (i.e., a domain expert, being a systems engineer in the case of the chosen use case) in the design and early validation & verification of a complex system. This scenario shows how to edit a model, and execute it such as the language user can interact with it to inject external stimuli through the event manager and drive the execution accordingly. In addition, the scenarios demonstrates the various facilities provided to help manually debugging the model (wrt. to an expected behavior), such as breakpoint definition and time traveling (forward and backward).

3.1. Model Edition

The Modeling workbench designed in Chapter 2 offers several model editors.

3.1.1. Textual Editor in Action

The smallest valid FCL model would be similar to Listing smallest FCL model. With a single mode and a main function that does nothing.

Listing smallest FCL model

```
FlightControllerModel SmallestController {  ①
    mainModeStateMachine {
        mode ManualAcrobaticFlight {}       ②
        initialMode ManualAcrobaticFlight   ③
    }
    mainFunction {                          ④
    }
}
```

① name of the model

② definition of a mode

③ definition of the initial mode

④ main function

A slighty more complex example would look like Listing Small FCL model with one master function. It defines

Listing Small FCL model with one master function

```
FlightControllerModel SmallController {
    mainModeStateMachine {
        mode ManualAcrobaticFlight {}
        initialMode ManualAcrobaticFlight
    }
    mainFunction {
        ports {
            dataPort in i : integer defaultValue 0  ①
            dataPort out o : integer
        }
        action {
            o := (i * 10)                           ②
        }
    }
    dataTypes {
        IntegerValueType integer {}                 ③
    }
}
```

① the master function exposes an input port `i` and an output port `o`

② the action indicates how to compute `o`

③ the model defines an integer type

And a minimal system that defines several modes will look like Listing Minimal FCL model with modes.

Listing Minimal FCL model with modes

```
FlightControllerModel minimalController {
    mainModeStateMachine {
        mode MultBy10 {
            enabledFunctions (X10)          ①
        }
        mode MultBy100 {
            enabledFunctions (X10)          ①
        }
        final mode Stop {}                  ②
        initialMode MultBy10

        transition MultBy10 -> MultBy100 { when buttonPressed } ③
        transition MultBy100 -> Stop { when buttonPressed }
    }
    mainFunction {
        ports {
            dataPort in i : integer defaultValue 0
            dataPort out o : integer
        }
        dataFlow {                          ④
```

```
            function X10 {
                ports {
                    dataPort in ix10 : integer defaultValue 0
                    dataPort out ox10 : integer
                }
                action {
                    ox10 := (ix10 * 10)     ⑤
                }
            }
            function X100 {
                ports {
                    dataPort in ix100 : integer defaultValue 0
                    dataPort out ox100 : integer
                }
                action {
                    ox100 := (ix100 * 100)  ⑤
                }
            }
            connect X10.ox10 <-> o          ⑥
            connect X100.ox100 <-> o
            connect i <-> X10.ix10
            connect i <-> X100.ix100
        }
    }
    events {
        external buttonPressed              ⑦
    }
    dataTypes {
        IntegerValueType integer {}
    }
}
```

① each mode indicates which sub function is active or not.

② some mode may be final to end the execution.

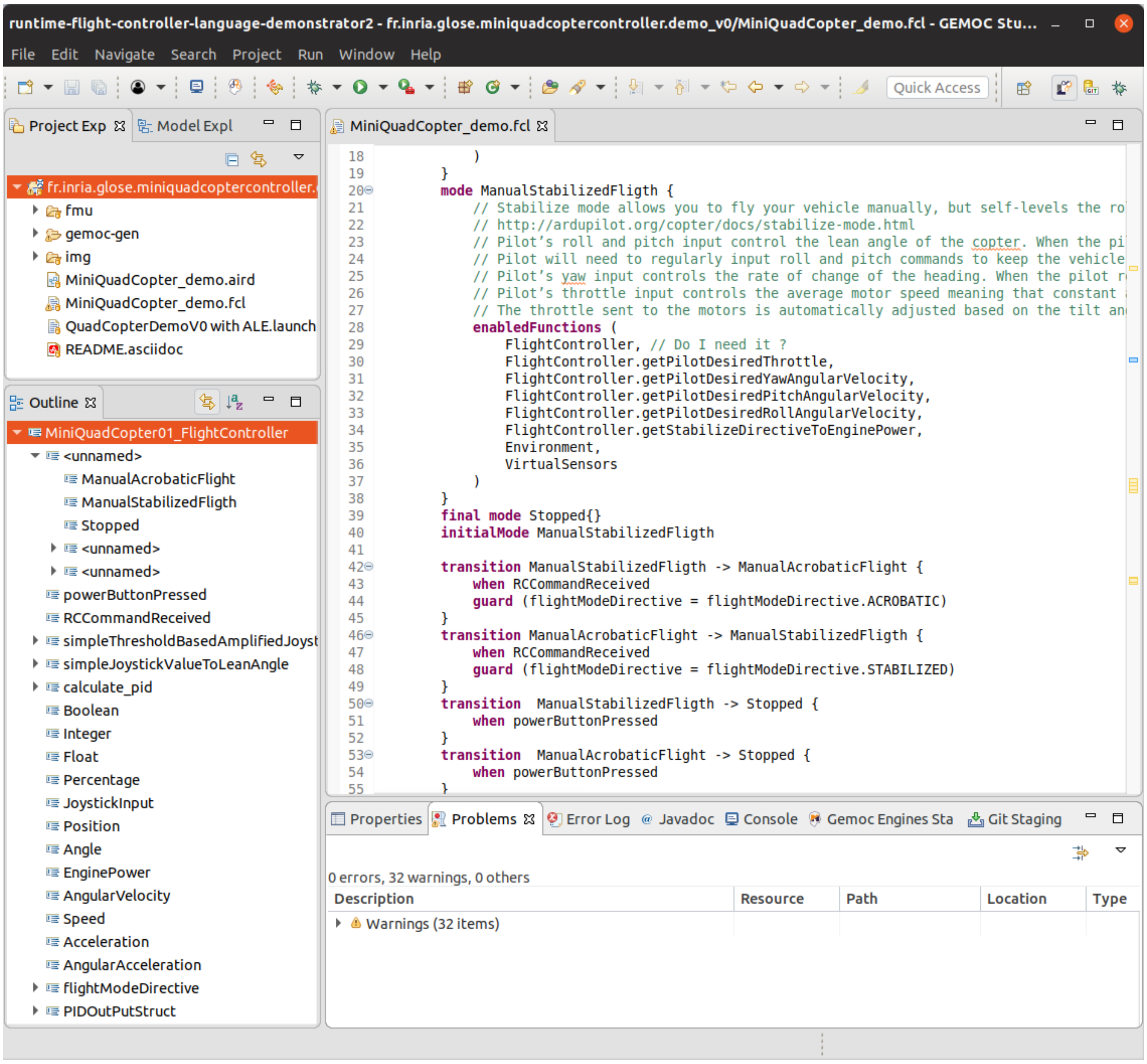
③ transitions between modes. (can be triggered via events and/or guard on data).

④ the main function declares 2 sub functions (in this case they are exclusive).

⑤ each sub function declare an action indicates how to compute its output.

⑥ mapping between ports (in this case between child functions to parent functions, but this can also be between 2 children functions).

⑦ an event that can be raised (either internally from an action or from the system environment).



```
            )
        }
        mode ManualStabilizedFligth {
            // Stabilize mode allows you to fly your vehicle manually, but self-levels the ro
            // http://ardupilot.org/copter/docs/stabilize-mode.html
            // Pilot’s roll and pitch input control the lean angle of the copter. When the pi
            // Pilot will need to regularly input roll and pitch commands to keep the vehicle
            // Pilot’s yaw input controls the rate of change of the heading. When the pilot r
            // Pilot’s throttle input controls the average motor speed meaning that constant
            // The throttle sent to the motors is automatically adjusted based on the tilt an
            enabledFunctions (
                FlightController, // Do I need it ?
                FlightController.getPilotDesiredThrottle,
                FlightController.getPilotDesiredYawAngularVelocity,
                FlightController.getPilotDesiredPitchAngularVelocity,
                FlightController.getPilotDesiredRollAngularVelocity,
                FlightController.getStabilizeDirectiveToEnginePower,
                Environment,
                VirtualSensors
            )
        }
        final mode Stopped{}
        initialMode ManualStabilizedFligth

        transition ManualStabilizedFligth -> ManualAcrobaticFlight {
            when RCCommandReceived
            guard (flightModeDirective = flightModeDirective.ACROBATIC)
        }
        transition ManualAcrobaticFlight -> ManualStabilizedFligth {
            when RCCommandReceived
            guard (flightModeDirective = flightModeDirective.STABILIZED)
        }
        transition  ManualStabilizedFligth -> Stopped {
            when powerButtonPressed
        }
        transition  ManualAcrobaticFlight -> Stopped {
            when powerButtonPressed
        }
```

0 errors, 32 warnings, 0 others

Description	Resource	Path	Location	Type
Warnings (32 items)				

Figure 13. Xtext editor opened on the demo controller (mode statemachine part)

3.1.2. Graphical Editor in Action

Opening the Sirius representations of the minimal controller Listing Minimal FCL model with modes provide the diagrams of Figure 14.

MinimalController-Sirius-editor-Screenshot.png

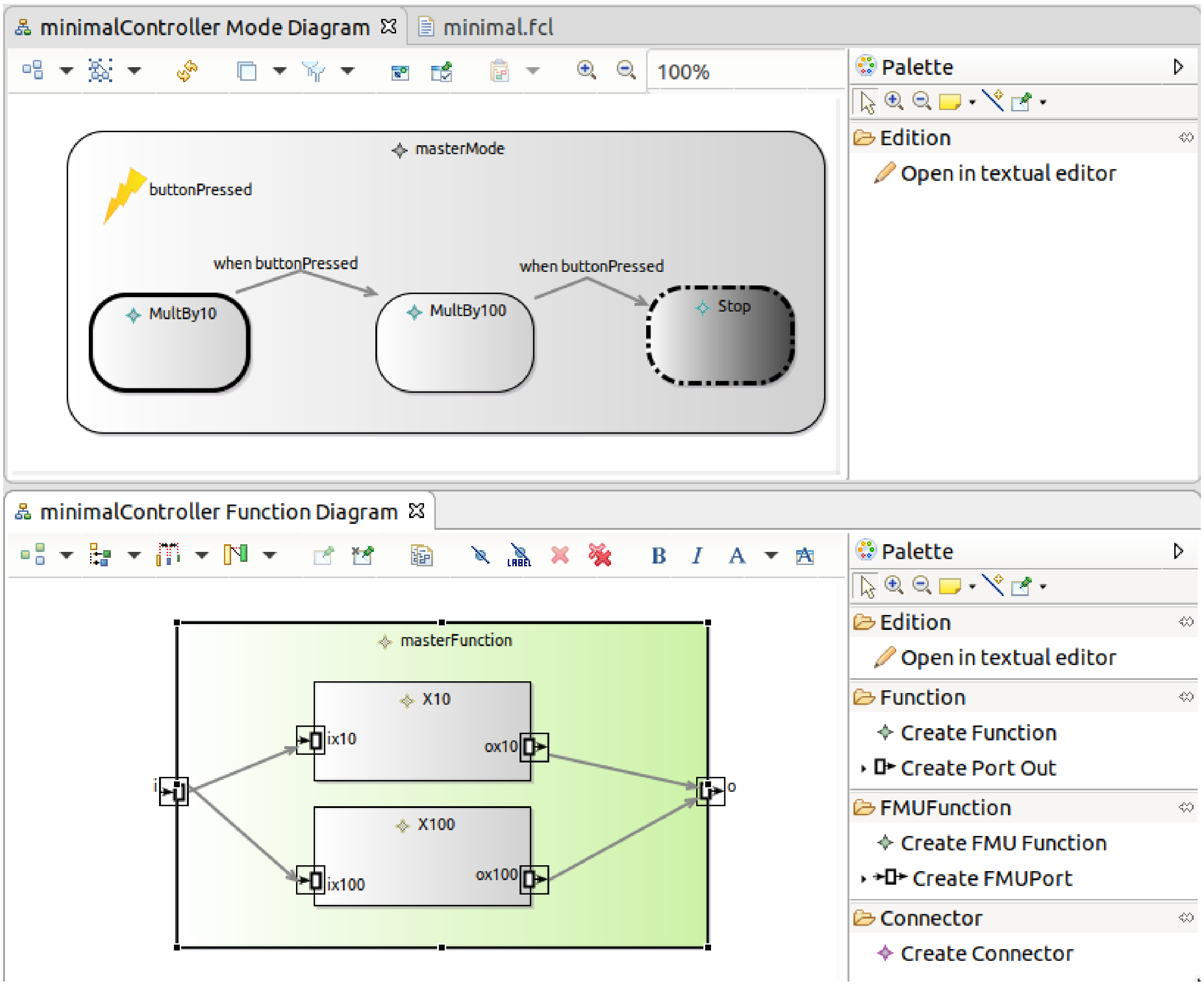


Figure 14. Small FCL model opened with the graphical editor.

TODO write section

Figure 12 and Figure 16 show the graphical editor diagrams applied to the Quadcopter demo controller.

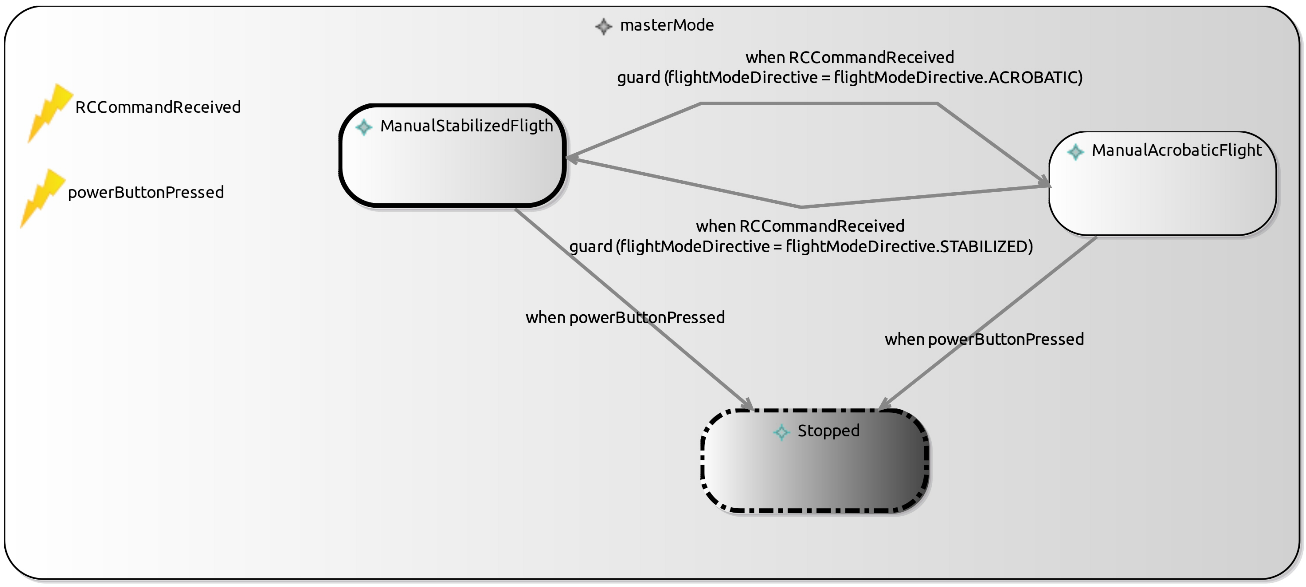


Figure 15. Mini quadcopter controller mode diagram

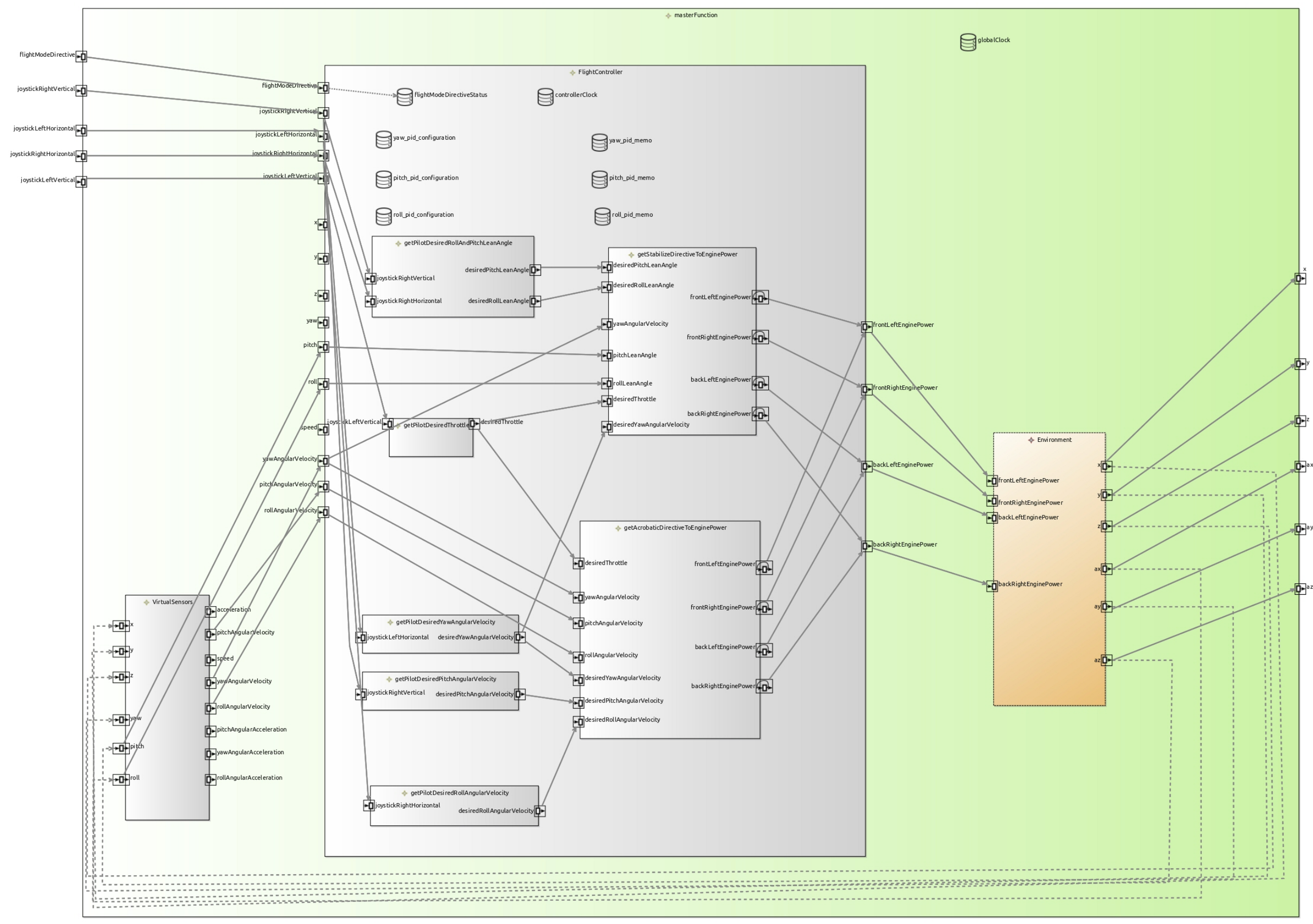


Figure 16. Mini quadcopter controller function diagram

3.1.3. Static Validation

The validation rules added in [sec-fcl-static-validation] are reported in the Modeling workbench in several ways:

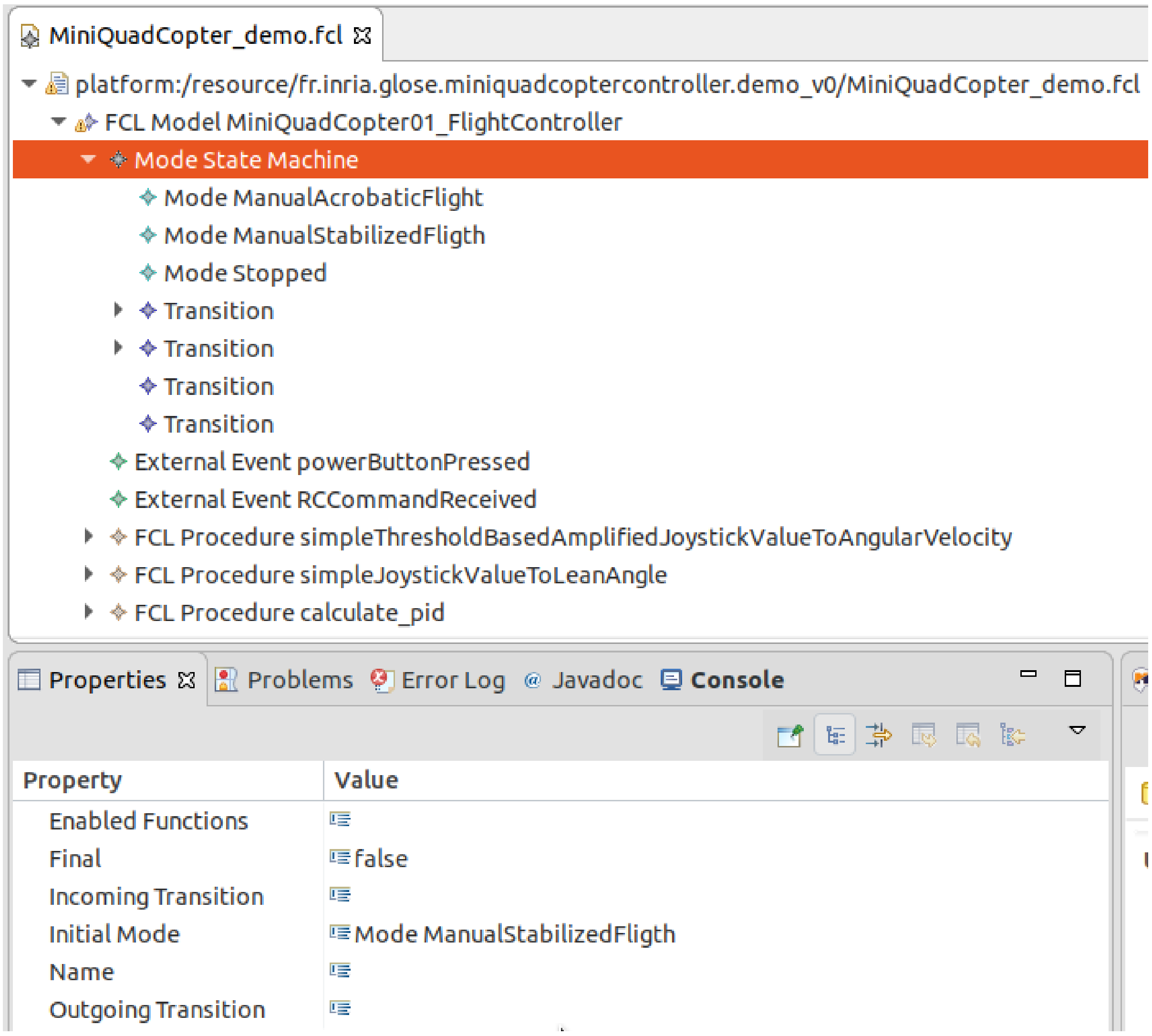
- directly in the editors using colored icons; (hovering over the element shows a tooltip with an explanatory text.)
- in a summary in the *Problem view*

TODO show some error and warning in the various views

3.1.4. Basic Tree Editor in Action

As seen in Section 2.1.2, a simple tree editor is also available. since we haven't generated the dedicated java code for it, the Modeler can open a fcl file using a *Right click on the .fcl file → Open With → Other → Sample Reflective Ecore Model Editor*

Figure 17 shows this tree editor on the basic quadcopter controller.



MiniQuadCopter_demo.fcl

platform:/resource/fr.inria.glose.miniquadcoptercontroller.demo_v0/MiniQuadCopter_demo.fcl
- FCL Model MiniQuadCopter01_FlightController
  - Mode State Machine
    - Mode ManualAcrobaticFlight
    - Mode ManualStabilizedFligth
    - Mode Stopped
    - Transition
    - Transition
    - Transition
    - Transition
  - External Event powerButtonPressed
  - External Event RCCommandReceived
  - FCL Procedure simpleThresholdBasedAmplifiedJoystickValueToAngularVelocity
  - FCL Procedure simpleJoystickValueToLeanAngle
  - FCL Procedure calculate_pid

Property	Value
Enabled Functions	
Final	false
Incoming Transition	
Initial Mode	Mode ManualStabilizedFligth
Name	
Outgoing Transition	

Figure 17. DemoController model in tree editor screenshot

3.2. Debugger in Action

Figure 18 shows the toggle breakpoint action and some elements with the *breakpoint* icon that is provided by the **Debug** layer.

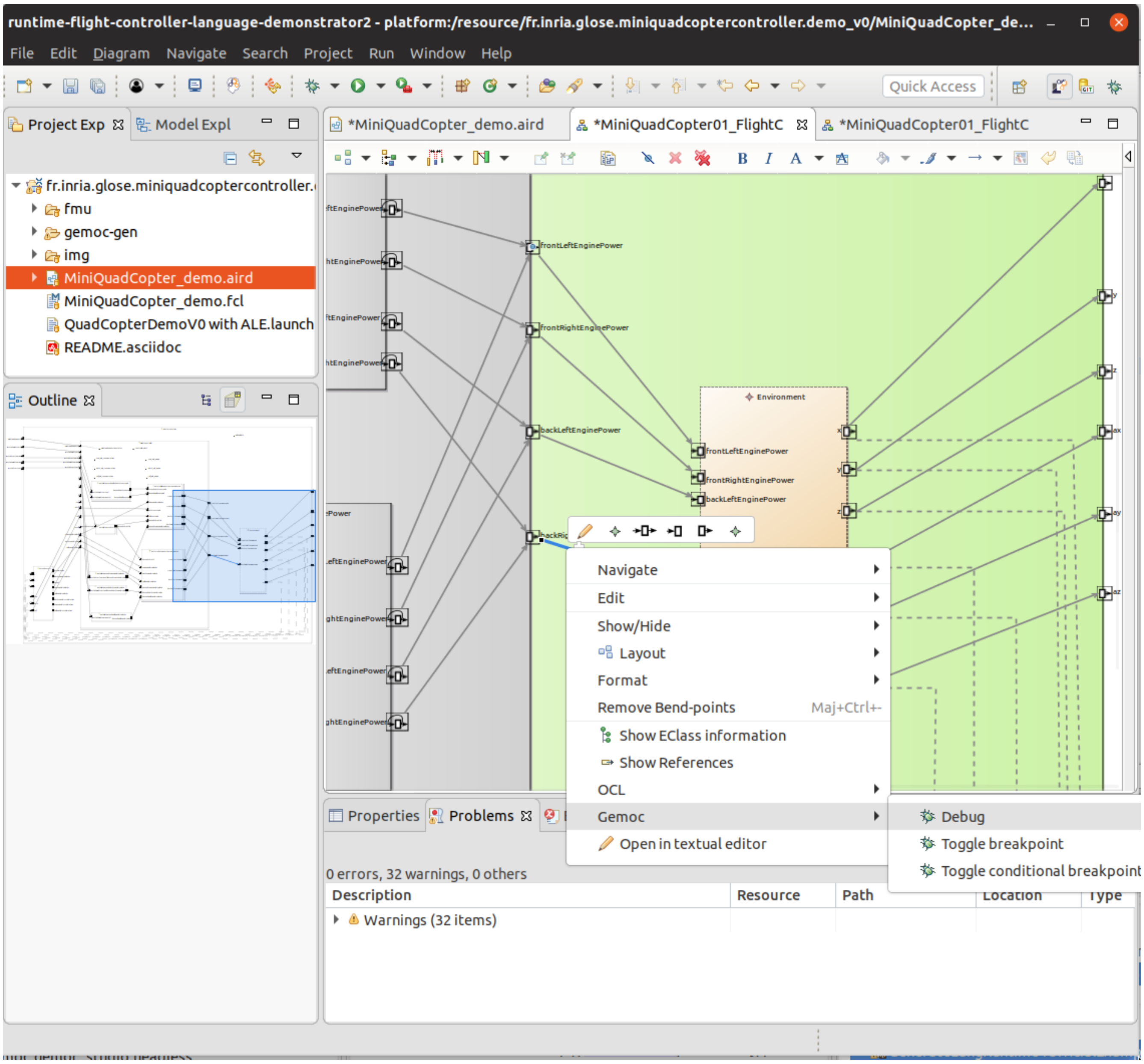
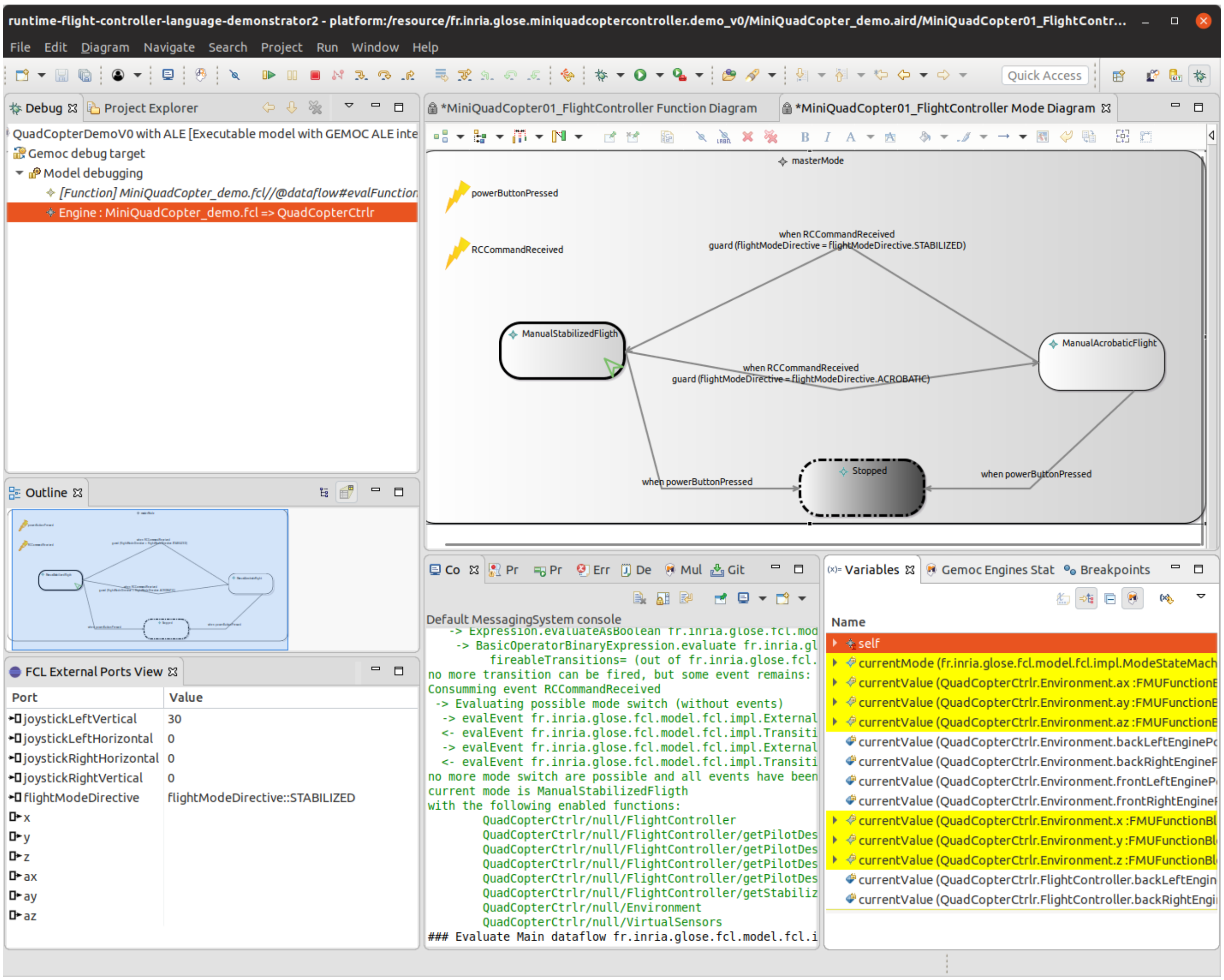

runtime-flight-controller-language-demonstrator2 - platform:/resource/fr.inria.glose.miniquadcoptercontroller.demo_v0/MiniQuadCopter_de...
File Edit Diagram Navigate Search Project Run Window Help
Quick Access
Project Exp
Model Expl
fr.inria.glose.miniquadcoptercontroller.
fmu
gemoc-gen
img
MiniQuadCopter_demo.aird
MiniQuadCopter_demo.fcl
QuadCopterDemoV0 with ALE.launch
README.asciidoc
Outline
*MiniQuadCopter_demo.aird
*MiniQuadCopter01_FlightC
*MiniQuadCopter01_FlightC
frontLeftEnginePower
frontRightEnginePower
backLeftEnginePower
Environment
Navigate
Edit
Show/Hide
Layout
Format
Remove Bend-points Maj+Ctrl+-
Show EClass information
Show References
OCL
Gemoc
Open in textual editor
Debug
Toggle breakpoint
Toggle conditional breakpoint
Properties
Problems
0 errors, 32 warnings, 0 others
Description
Resource
Path
Location
Type
Warnings (32 items)

Figure 18. *Toggle breakpoint in action on the DataFlow diagram.*

3.3. Animation in Action


runtime-flight-controller-language-demonstrator2 - platform:/resource/fr.inria.glose.miniquadcoptercontroller.demo_v0/MiniQuadCopter_demo.aird/MiniQuadCopter01_FlightContr...
File Edit Diagram Navigate Search Project Run Window Help
Quick Access
Debug
Project Explorer
*MiniQuadCopter01_FlightController Function Diagram
*MiniQuadCopter01_FlightController Mode Diagram
masterMode
powerButtonPressed
RCCommandReceived
when RCCommandReceived
guard (flightModeDirective = flightModeDirective.STABILIZED)
ManualStabilizedFligth
ManualAcrobaticFlight
when RCCommandReceived
guard (flightModeDirective = flightModeDirective.ACROBATIC)
Stopped
when powerButtonPressed
when powerButtonPressed
Outline
FCL External Ports View
Co Pr Pr Err De Mul Git
Variables
Gemoc Engines Stat
Breakpoints

QuadCopterDemoV0 with ALE [Executable model with GEMOC ALE inte
- Gemoc debug target
  - Model debugging
    - *[Function] MiniQuadCopter_demo.fcl//@dataflow#evalFunction*
    - Engine : MiniQuadCopter_demo.fcl => QuadCopterCtrlr

Port	Value
joystickLeftVertical	30
joystickLeftHorizontal	0
joystickRightHorizontal	0
joystickRightVertical	0
flightModeDirective	flightModeDirective::STABILIZED
x	
y	
z	
ax	
ay	
az	

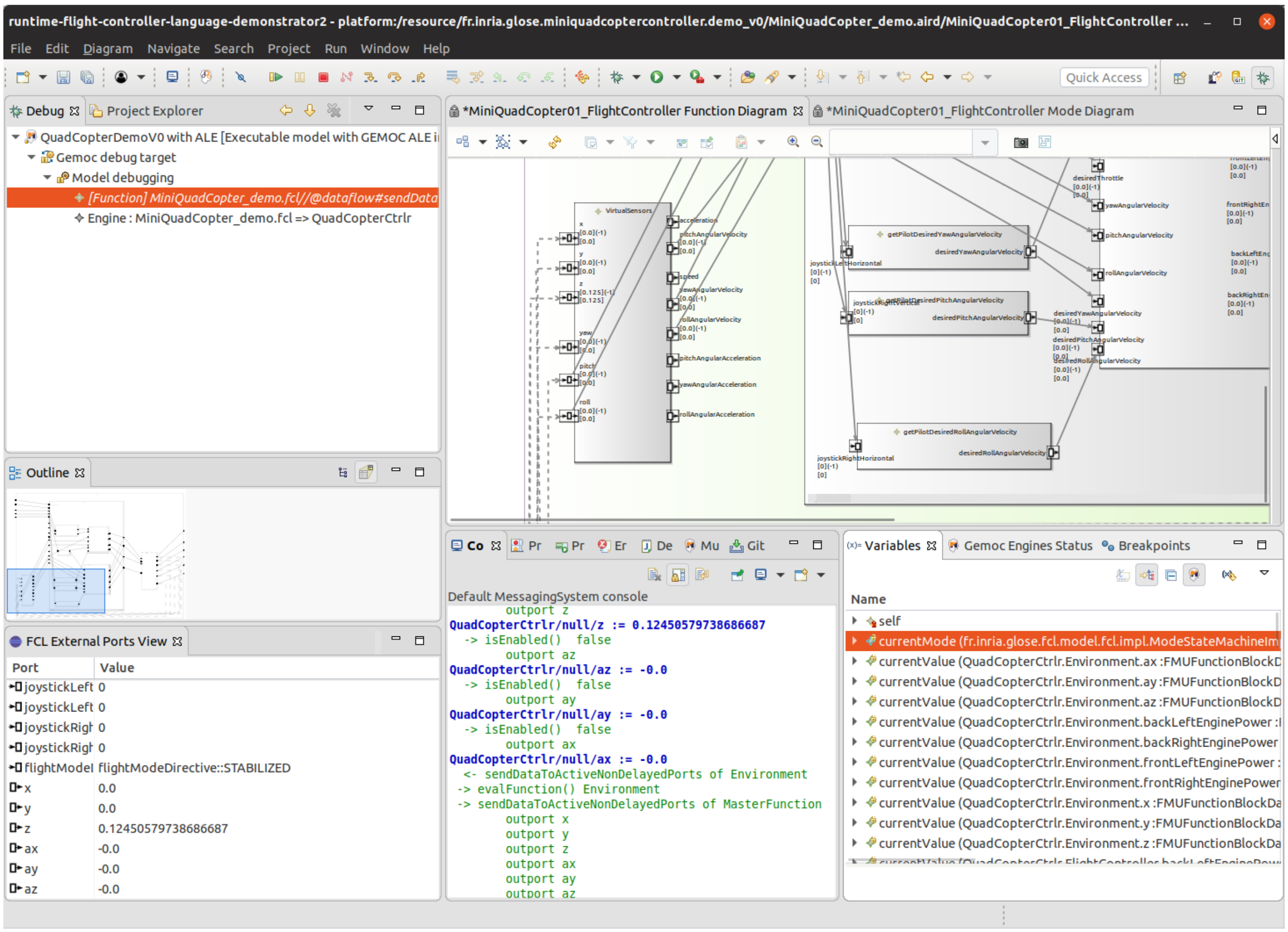
Default MessagingSystem console

```
   -> Expression.evaluateAsBoolean fr.inria.glose.fcl.mod
    -> BasicOperatorBinaryExpression.evaluate fr.inria.gl
        fireableTransitions= (out of fr.inria.glose.fcl.
no more transition can be fired, but some event remains:
Consumming event RCCommandReceived
 -> Evaluating possible mode switch (without events)
  -> evalEvent fr.inria.glose.fcl.model.fcl.impl.External
  <- evalEvent fr.inria.glose.fcl.model.fcl.impl.Transiti
  -> evalEvent fr.inria.glose.fcl.model.fcl.impl.External
  <- evalEvent fr.inria.glose.fcl.model.fcl.impl.Transiti
no more mode switch are possible and all events have been
current mode is ManualStabilizedFligth
with the following enabled functions:
        QuadCopterCtrlr/null/FlightController
        QuadCopterCtrlr/null/FlightController/getPilotDes
        QuadCopterCtrlr/null/FlightController/getPilotDes
        QuadCopterCtrlr/null/FlightController/getPilotDes
        QuadCopterCtrlr/null/FlightController/getPilotDes
        QuadCopterCtrlr/null/FlightController/getStabiliz
        QuadCopterCtrlr/null/Environment
        QuadCopterCtrlr/null/VirtualSensors
### Evaluate Main dataflow fr.inria.glose.fcl.model.fcl.i
```

Name
self
currentMode (fr.inria.glose.fcl.model.fcl.impl.ModeStateMach
currentValue (QuadCopterCtrlr.Environment.ax :FMUFunctionE
currentValue (QuadCopterCtrlr.Environment.ay :FMUFunctionE
currentValue (QuadCopterCtrlr.Environment.az :FMUFunctionE
currentValue (QuadCopterCtrlr.Environment.backLeftEnginePo
currentValue (QuadCopterCtrlr.Environment.backRightEngineP
currentValue (QuadCopterCtrlr.Environment.frontLeftEngineP
currentValue (QuadCopterCtrlr.Environment.frontRightEngineI
currentValue (QuadCopterCtrlr.Environment.x :FMUFunctionBl
currentValue (QuadCopterCtrlr.Environment.y :FMUFunctionBl
currentValue (QuadCopterCtrlr.Environment.z :FMUFunctionBl
currentValue (QuadCopterCtrlr.FlightController.backLeftEngin
currentValue (QuadCopterCtrlr.FlightController.backRightEngi

Figure 19. Demo controller model running and paused after having changed mode.

Figure 20 shows the the demo controller model running and paused having evaluated all functions one time. The *FCL External Ports view* displays the current value of the port located on the MasterFunction (ie. the outer function)


runtime-flight-controller-language-demonstrator2 - platform:/resource/fr.inria.glose.miniquadcoptercontroller.demo_v0/MiniQuadCopter_demo.aird/MiniQuadCopter01_FlightController ...
File Edit Diagram Navigate Search Project Run Window Help
Debug
Project Explorer
*MiniQuadCopter01_FlightController Function Diagram
*MiniQuadCopter01_FlightController Mode Diagram
Outline
Variables
Gemoc Engines Status
Breakpoints

QuadCopterDemoV0 with ALE [Executable model with GEMOC ALE i
Gemoc debug target
Model debugging
[Function] MiniQuadCopter_demo.fcl//@dataflow#sendData
Engine : MiniQuadCopter_demo.fcl => QuadCopterCtrlr

FCL External Ports View

Port	Value
joystickLeft	0
joystickLeft	0
joystickRigh	0
joystickRigh	0
flightModel	flightModeDirective::STABILIZED
x	0.0
y	0.0
z	0.12450579738686687
ax	-0.0
ay	-0.0
az	-0.0

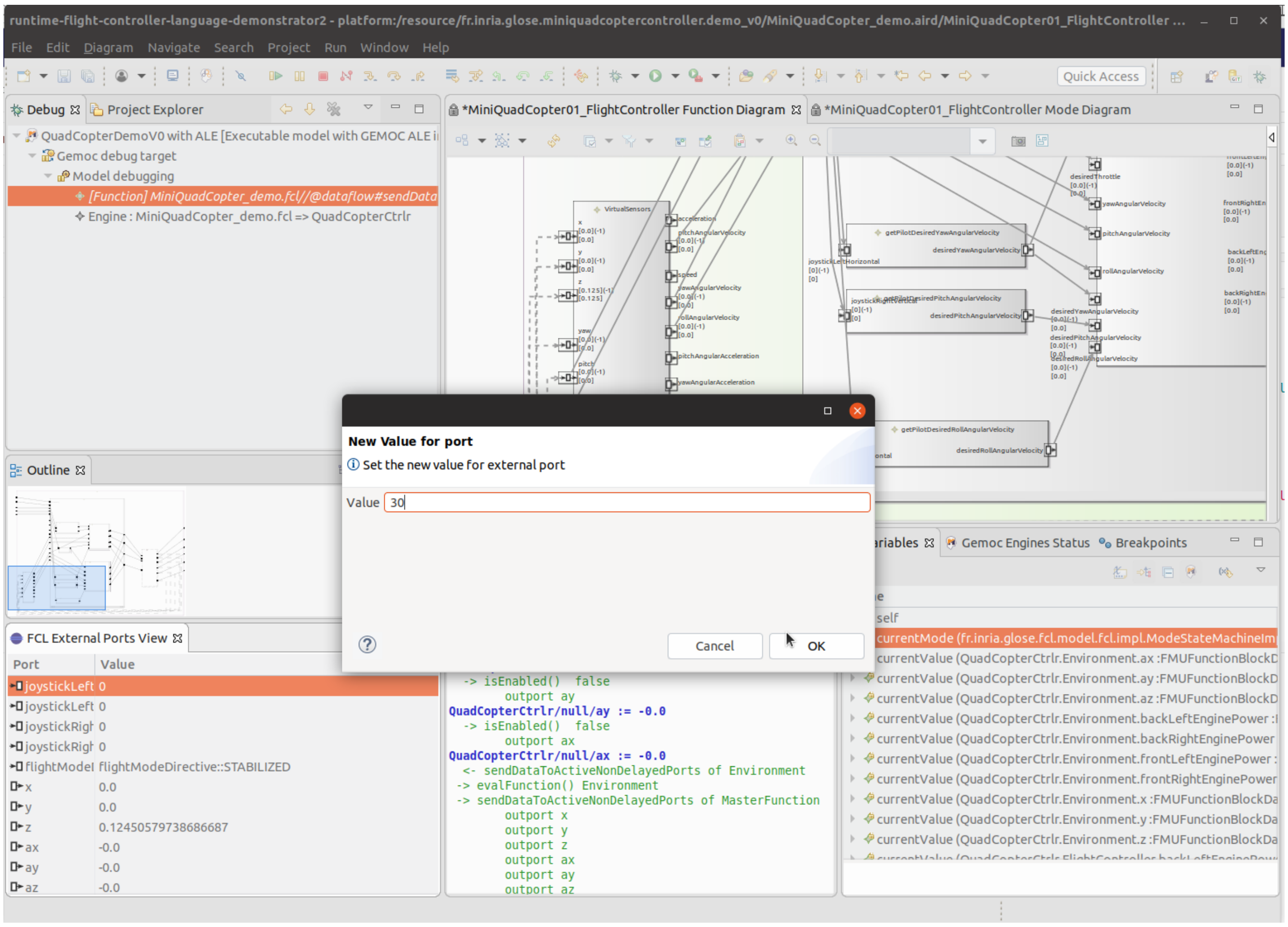
Default MessagingSystem console

```
         outport z
QuadCopterCtrlr/null/z := 0.12450579738686687
  -> isEnabled()  false
         outport az
QuadCopterCtrlr/null/az := -0.0
  -> isEnabled()  false
         outport ay
QuadCopterCtrlr/null/ay := -0.0
  -> isEnabled()  false
         outport ax
QuadCopterCtrlr/null/ax := -0.0
  <- sendDataToActiveNonDelayedPorts of Environment
 -> evalFunction() Environment
 -> sendDataToActiveNonDelayedPorts of MasterFunction
         outport x
         outport y
         outport z
         outport ax
         outport ay
         outport az
```

Name
- self
- currentMode (fr.inria.glose.fcl.model.fcl.impl.ModeStateMachineIm
- currentValue (QuadCopterCtrlr.Environment.ax :FMUFunctionBlockD
- currentValue (QuadCopterCtrlr.Environment.ay :FMUFunctionBlockD
- currentValue (QuadCopterCtrlr.Environment.az :FMUFunctionBlockD
- currentValue (QuadCopterCtrlr.Environment.backLeftEnginePower :
- currentValue (QuadCopterCtrlr.Environment.backRightEnginePower
- currentValue (QuadCopterCtrlr.Environment.frontLeftEnginePower :
- currentValue (QuadCopterCtrlr.Environment.frontRightEnginePower
- currentValue (QuadCopterCtrlr.Environment.x :FMUFunctionBlockDa
- currentValue (QuadCopterCtrlr.Environment.y :FMUFunctionBlockDa
- currentValue (QuadCopterCtrlr.Environment.z :FMUFunctionBlockDa

Figure 20. Demo controller model running and paused after having evaluated all functions one time.

Figure 21 shows how a double click (or right click → popup menu → *Change Value*) in the *FCL External Ports view* allows to changes its value.


runtime-flight-controller-language-demonstrator2 - platform:/resource/fr.inria.glose.miniquadcoptercontroller.demo_v0/MiniQuadCopter_demo.aird/MiniQuadCopter01_FlightController ...

QuadCopterDemoV0 with ALE [Executable model with GEMOC ALE i
Gemoc debug target
Model debugging
[Function] MiniQuadCopter_demo.fcl//@dataflow#sendData
Engine : MiniQuadCopter_demo.fcl => QuadCopterCtrlr

New Value for port
Set the new value for external port
Value 30

FCL External Ports View

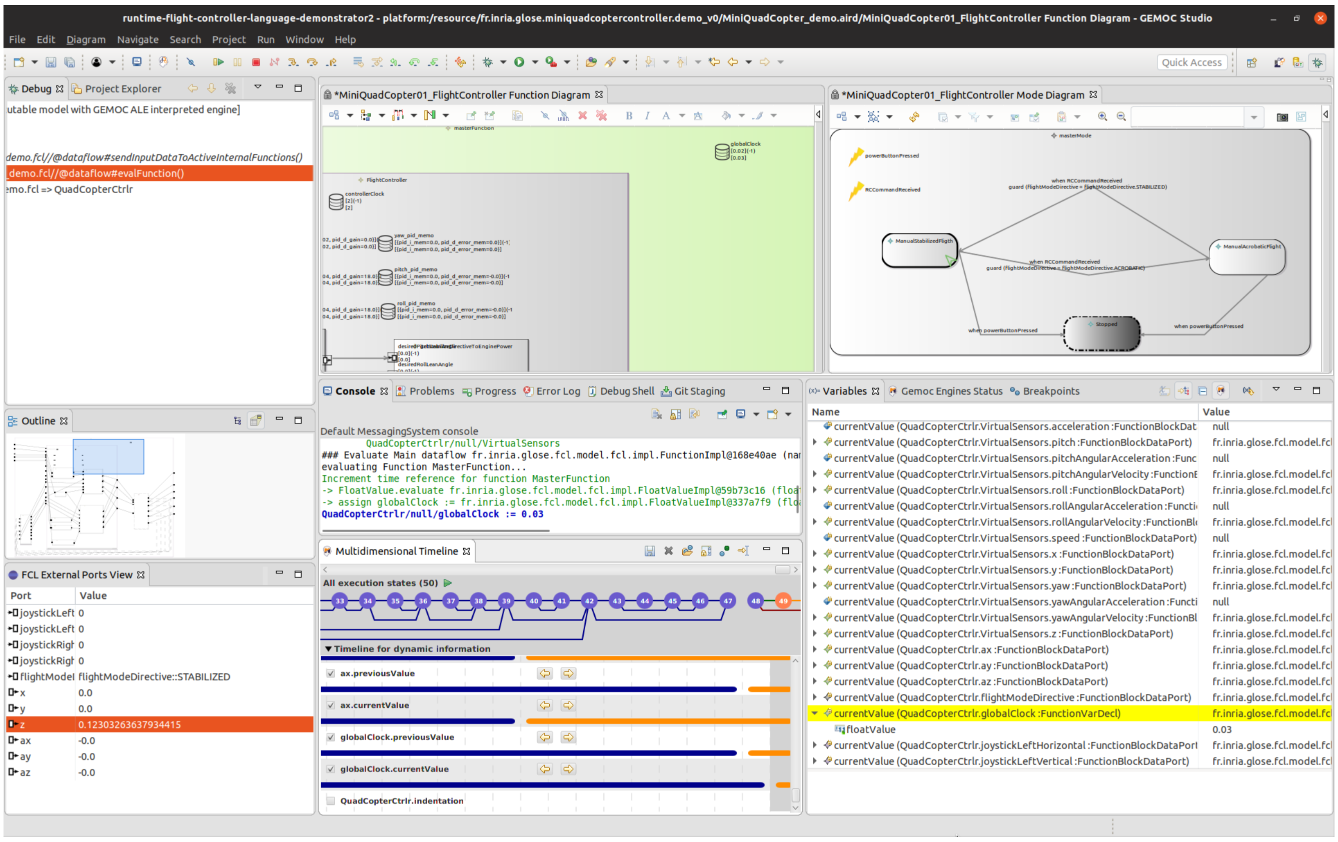
Port	Value
joystickLeft	0
joystickLeft	0
joystickRigh	0
joystickRigh	0
flightModeI	flightModeDirective::STABILIZED
x	0.0
y	0.0
z	0.12450579738686687
ax	-0.0
ay	-0.0
az	-0.0

```
  -> isEnabled()  false
        outport ay
QuadCopterCtrlr/null/ay := -0.0
  -> isEnabled()  false
        outport ax
QuadCopterCtrlr/null/ax := -0.0
  <- sendDataToActiveNonDelayedPorts of Environment
 -> evalFunction() Environment
 -> sendDataToActiveNonDelayedPorts of MasterFunction
        outport x
        outport y
        outport z
        outport ax
        outport ay
        outport az
```

```
self
currentMode (fr.inria.glose.fcl.model.fcl.impl.ModeStateMachineIm
currentValue (QuadCopterCtrlr.Environment.ax :FMUFunctionBlockD
currentValue (QuadCopterCtrlr.Environment.ay :FMUFunctionBlockD
currentValue (QuadCopterCtrlr.Environment.az :FMUFunctionBlockD
currentValue (QuadCopterCtrlr.Environment.backLeftEnginePower :I
currentValue (QuadCopterCtrlr.Environment.backRightEnginePower
currentValue (QuadCopterCtrlr.Environment.frontLeftEnginePower :
currentValue (QuadCopterCtrlr.Environment.frontRightEnginePower
currentValue (QuadCopterCtrlr.Environment.x :FMUFunctionBlockDa
currentValue (QuadCopterCtrlr.Environment.y :FMUFunctionBlockDa
currentValue (QuadCopterCtrlr.Environment.z :FMUFunctionBlockDa
```

Figure 21. Demo controller changing value using the ExternalPorts view.

Figure 22 shows the Demo controller paused after an update of the global clock. The Multidimensional timeline shows the changes on the current value. The value is also visible in the Variables view and the Sirius diagram.


runtime-flight-controller-language-demonstrator2 - platform:/resource/fr.inria.glose.miniquadcoptercontroller.demo_v0/MiniQuadCopter_demo.aird/MiniQuadCopter01_FlightController Function Diagram - GEMOC Studio

```
utable model with GEMOC ALE interpreted engine]

demo.fcl//@dataflow#sendInputDataToActiveInternalFunctions()
_demo.fcl//@dataflow#evalFunction()
emo.fcl => QuadCopterCtrlr
```

Default MessagingSystem console

```
        QuadCopterCtrlr/null/VirtualSensors
### Evaluate Main dataflow fr.inria.glose.fcl.model.fcl.impl.FunctionImpl@168e40ae (nar
evaluating Function MasterFunction...
Increment time reference for function MasterFunction
-> FloatValue.evaluate fr.inria.glose.fcl.model.fcl.impl.FloatValueImpl@59b73c16 (floa
-> assign globalClock := fr.inria.glose.fcl.model.fcl.impl.FloatValueImpl@337a7f9 (flo
QuadCopterCtrlr/null/globalClock := 0.03
```

FCL External Ports View

Port	Value
joystickLeft	0
joystickLeft	0
joystickRigh	0
joystickRigh	0
flightModel	flightModeDirective::STABILIZED
x	0.0
y	0.0
z	0.12303263637934415
ax	-0.0
ay	-0.0
az	-0.0

Variables

Name	Value
currentValue (QuadCopterCtrlr.VirtualSensors.acceleration :FunctionBlockDat	null
currentValue (QuadCopterCtrlr.VirtualSensors.pitch :FunctionBlockDataPort)	fr.inria.glose.fcl.model.fcl
currentValue (QuadCopterCtrlr.VirtualSensors.pitchAngularAcceleration :Func	null
currentValue (QuadCopterCtrlr.VirtualSensors.pitchAngularVelocity :FunctionE	fr.inria.glose.fcl.model.fcl
currentValue (QuadCopterCtrlr.VirtualSensors.roll :FunctionBlockDataPort)	fr.inria.glose.fcl.model.fcl
currentValue (QuadCopterCtrlr.VirtualSensors.rollAngularAcceleration :Functi	null
currentValue (QuadCopterCtrlr.VirtualSensors.rollAngularVelocity :FunctionBl	fr.inria.glose.fcl.model.fcl
currentValue (QuadCopterCtrlr.VirtualSensors.speed :FunctionBlockDataPort)	null
currentValue (QuadCopterCtrlr.VirtualSensors.x :FunctionBlockDataPort)	fr.inria.glose.fcl.model.fcl
currentValue (QuadCopterCtrlr.VirtualSensors.y :FunctionBlockDataPort)	fr.inria.glose.fcl.model.fcl
currentValue (QuadCopterCtrlr.VirtualSensors.yaw :FunctionBlockDataPort)	fr.inria.glose.fcl.model.fcl
currentValue (QuadCopterCtrlr.VirtualSensors.yawAngularAcceleration :Functi	null
currentValue (QuadCopterCtrlr.VirtualSensors.yawAngularVelocity :FunctionBl	fr.inria.glose.fcl.model.fcl
currentValue (QuadCopterCtrlr.VirtualSensors.z :FunctionBlockDataPort)	fr.inria.glose.fcl.model.fcl
currentValue (QuadCopterCtrlr.ax :FunctionBlockDataPort)	fr.inria.glose.fcl.model.fcl
currentValue (QuadCopterCtrlr.ay :FunctionBlockDataPort)	fr.inria.glose.fcl.model.fcl
currentValue (QuadCopterCtrlr.az :FunctionBlockDataPort)	fr.inria.glose.fcl.model.fcl
currentValue (QuadCopterCtrlr.flightModeDirective :FunctionBlockDataPort)	fr.inria.glose.fcl.model.fcl
currentValue (QuadCopterCtrlr.globalClock :FunctionVarDecl)	fr.inria.glose.fcl.model.fcl
floatValue	0.03
currentValue (QuadCopterCtrlr.joystickLeftHorizontal :FunctionBlockDataPort	fr.inria.glose.fcl.model.fcl
currentValue (QuadCopterCtrlr.joystickLeftVertical :FunctionBlockDataPort)	fr.inria.glose.fcl.model.fcl

Figure 22. Demo controller paused after an update of the global clock.

Chapter 4. Planning for V2

The version 1 of the demonstrator will be enhanced into a V2.

Most of them are listed in the gitlab: https://gitlab.inria.fr/glose/flight-controller-language-demonstrator/issues

4.1. Enhancement of Existing Features

4.2. New Features

- Concurrent Engine
- Scenario #3: Exploring the Design Space of a Behavioral Model This scenario will demonstrate the ability of the GEMOC Studio to explore the possible execution paths allowed by the model. As an outcome of such a scenario, the demonstrator will show how to record external stimuli as generic scenarios and reuse them as test cases.