GLOSE Demonstrator - DP1.1

Modeling environment for Flight Control Management

Table of Contents

1. Flight Controller Language - FCL . . . 3
1.1. Language Overview . . . 3
1.2. Repository Content Summary . . . 5
1.2.1. FCL Specifications projects . . . 5
1.2.2. Demonstration Controllers . . . 6
2. Scenario #1: Building an Executable Modeling Language . . . 7
2.1. FCL Metamodel Design . . . 7
2.1.1. FCL Ecore . . . 7
2.1.2. Basic Tree Editor . . . 12
2.2. Editors Design . . . 13
2.2.1. Xtext Editor . . . 13
2.2.2. Sirius Editor . . . 15
2.2.3. Sirius/Xtext Integration . . . 18
2.2.4. Static Validation . . . 18
2.3. Behavioral Semantic . . . 19
2.4. Deployment . . . 27
2.4.1. Packaging and Deployment in an Eclipse Product . . . 27
2.4.2. Runtime Deployment . . . 29
3. Scenario #2: Editing Model . . . 30
3.1. Model Edition . . . 30
3.1.1. Textual Editor in Action . . . 30
3.1.2. Graphical Editor in Action . . . 33
3.1.3. Static Validation . . . 34
3.1.4. Basic Tree Editor in Action . . . 35
4. Additional features scheduled for T0+12 . . . 37
4.1. Enhancement of Existing Features . . . 37
4.2. New Features . . . 37

Project Title	GLOSE Demonstrator
Deliverable Type	Software and document
Deliverable Number	DP1.1
Title of Deliverable	Modeling environment for Flight Control Management
Version	1.0
Contractual Delivery Date	T0+6, Sep. 2019
Author	Didier Vojtisek <didier.vojtisek@inria.fr>
Reviewer	Benoit Combemale <benoit.combemale@irisa.fr>
Latest version	https://gitlab.inria.fr/glose/flight-controller-language-demonstrator/tree/master/documentation

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

The version used while producing this document is available at https://gitlab.inria.fr/glose/flight-controller-language-demonstrator/-/tags/D1.1-v1.0.

The latest (possibly updated) online version of this document is also available as html or as pdf

This document presents the demonstrator illustrating the technologies involved in the WP2 of the GLOSE project and the associated use case language.

The demonstrator provides generals guidelines about how to use GEMOC technologies in order to design and implement a modeling language and its corresponding modeling environment.

The modeling environment provided by the demonstrator covers several concerns such as:

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

This document: DP1.1: *Modeling environment for Flight Control Management*, focuses on how to develop and use a modeling environment. The document DP1.2: *Simulation and Compilation Environment for Flight Control Management* will focus on model execution, model debugging, model animation, and compilation. The document DP2.1: *Execution and Simulation Environment for Flight Control Management with concurrent execution* will extend DP1.2 with concurrency concerns. The document DP3.1: *Source model for the case studies* will present example Flight Control Language models.

The demonstrator 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 modeling environment for a language involves two mains roles: language engineers and modelers.

- Language engineers design executable DSLs using metalanguages and implement modeling environments.
- Modelers are the users of the resulting modeling environment. They build models using the modeling language,and they analyze and execute them using the other tools of the environment.

The GEMOC Studio used to develop the 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 workbenches 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 modeling environment.

For FCL, one can use the following update site to add FCL modeling environment 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 the specification of the logic for such code.

An end user of such language (i.e. modeler) would then have a higher level of abstraction and better editing support allowing her to be more efficient when writing complex logic.

The Flight Controller Language (FCL) developed for this demonstrator is intended to specify relatively small functions (which might be complex in terms of control command laws on the rotors and sensors) and basic plans. These functions communicate data between them in order to create a data flow.

In order to adapt the behavior of the system to different situations, it uses a mechanism of modes. The modes filter the data flow functions that are enabled or not. Switching from a mode to another is driven by an automata.

For instance, typical basic modes that would be defined for a quadcopter would be inspired by the ones defined in ardupilot flight controller http://ardupilot.org/copter/docs/flight-modes.html i.e. a Stabilized mode (Self-levels the roll and pitch axis), Acrobatic mode (Holds attitude, no self-level), Land (Reduces altitude to ground level, attempts to go straight down), etc.

Figure 1 shows an example of mode automata with *ManualStabilizeFlight* mode as initial mode, *ManualAcrobaticFlight* as a second mode and a final *Stopped* mode.

Figure 2 shows an example data flow with various functions. Some of these functions are enabled only in some modes as shown in Listing Mini quadcopter controller enabled functions (excerpt of MiniQuadCopter_demo.fcl).

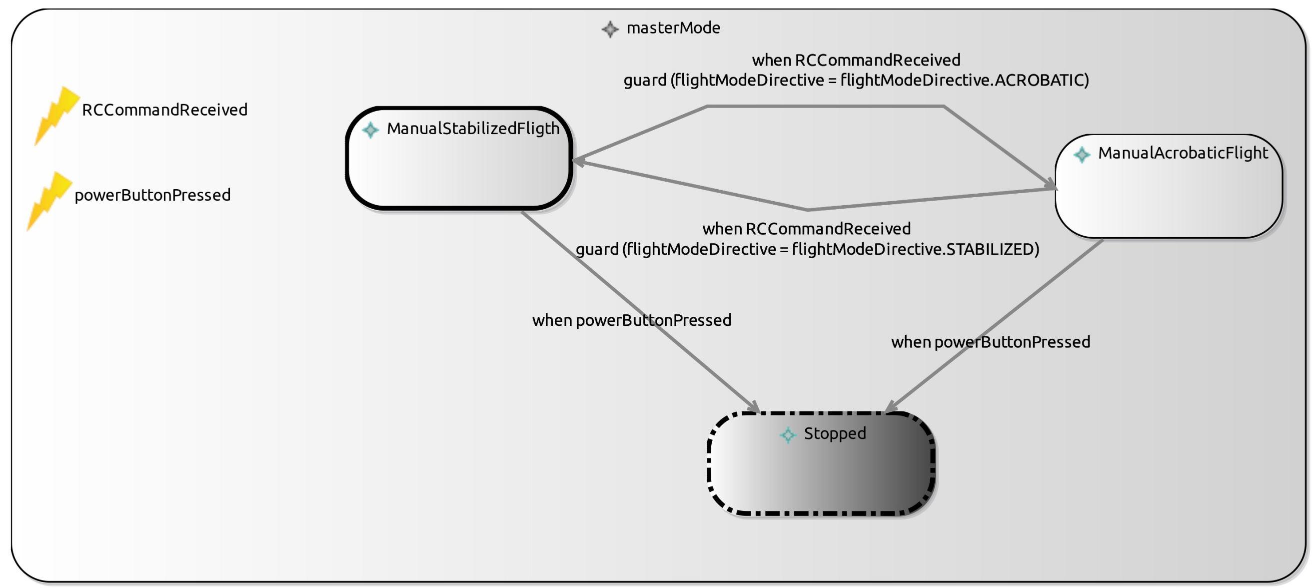

Figure 1. Mini quadcopter controller mode diagram

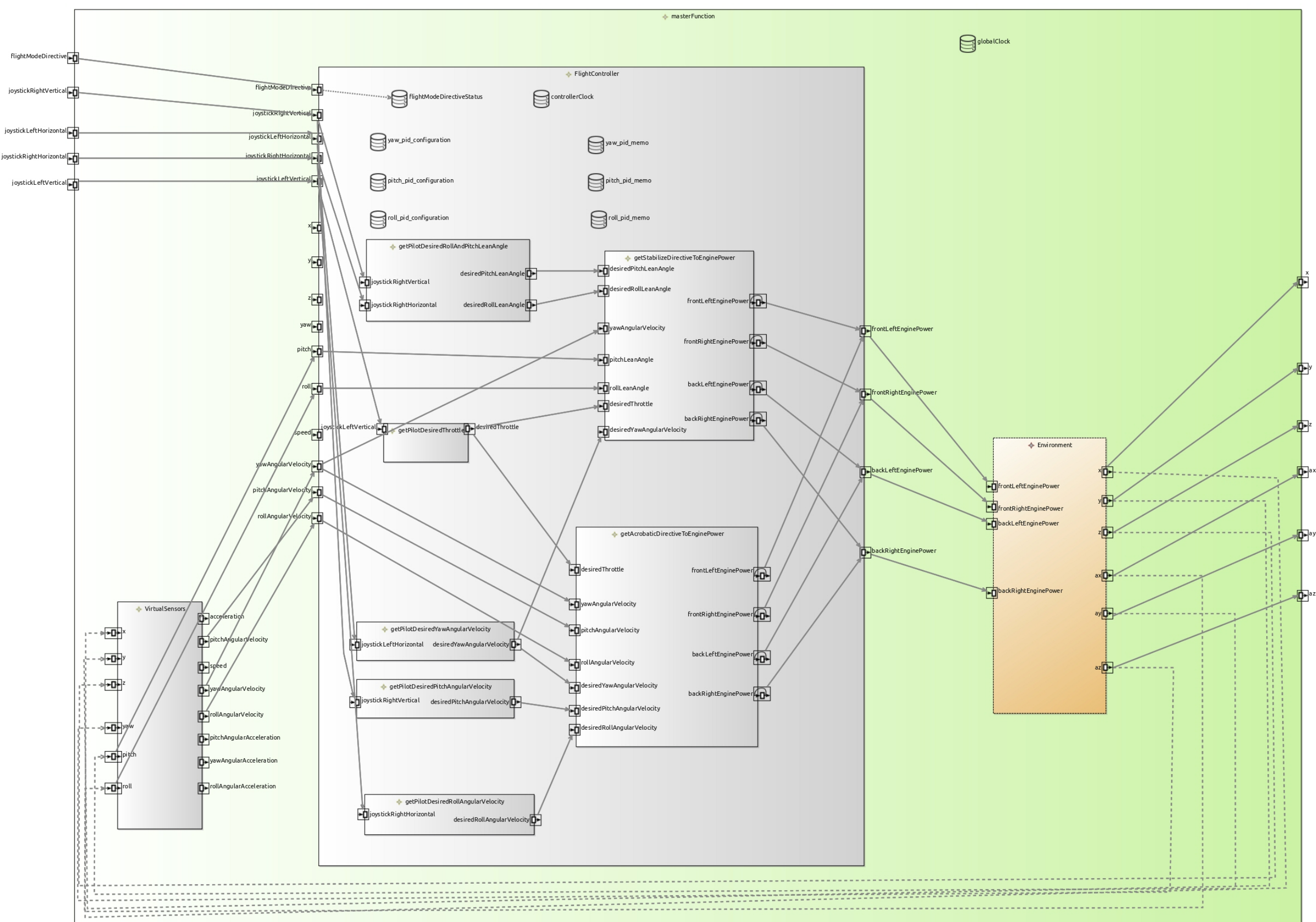

Figure 2. Mini quadcopter controller function diagram

Listing Mini quadcopter controller enabled functions (excerpt of MiniQuadCopter_demo.fcl)

```
FlightControllerModel QuadCopterCtrlr {
    mainModeStateMachine {
        mode ManualAcrobaticFlight {
            enabledFunctions (
                FlightController,
                FlightController.getPilotDesiredThrottle,
                FlightController.getPilotDesiredYawAngularVelocity,
                FlightController.getPilotDesiredRollAndPitchLeanAngle,
                FlightController.getAcrobaticDirectiveToEnginePower,
                Environment,
                VirtualSensors
            )
        }
        mode ManualStabilizedFligth {
            enabledFunctions (
                FlightController,
                FlightController.getPilotDesiredThrottle,
                FlightController.getPilotDesiredYawAngularVelocity,
                FlightController.getPilotDesiredPitchAngularVelocity,
                FlightController.getPilotDesiredRollAngularVelocity,
                FlightController.getStabilizeDirectiveToEnginePower,
                Environment,
                VirtualSensors
            )
        }
        final mode Stopped{}
        initialMode ManualStabilizedFligth
[...]
```

1.2. Repository Content Summary

The sources in git contains the elements decrisbed in the following subsections:

1.2.1. FCL Specifications projects

The specifications of FCL are in fcl-implementations.

The folder `fcl` contains the projects building a language that is not executable yet. This is a base for the behavior implementations (both for the ALEFCL implementation that is sequential and for the ConcurrentFCL that will be developed for D2.1)

- 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 the 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 contain 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 Ecore.
- 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.

The folder `releng` contains projects used for release engineering. i.e. specification about how to package and deploy the FCL language in an Eclipse installation. (cf. Section 2.4)

- releng/fr.inria.glose.fcl.all.feature project contains the list of Eclipse plugins that must be deployed.
- releng/fr.inria.glose.fcl.updatesite project contains the list of features that are published on the web and available for installation from an Eclipse IDE.
- releng/fr.inria.glose.fcl.target-platform project contains the definition of a typical target Eclipse IDE. It is used by a continuous integration server.

1.2.2. Demonstration Controllers

Some FCL model examples are located in the model_examples folder.

The demonstrator's 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 as shown in Figure 1 and Figure 2.

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

This scenario demonstrates the ability of the GEMOC Studio to help a language engineer 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.

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

2.1. FCL Metamodel Design

During this phase, the Language engineer use Ecore Tools to capture 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 relationships.

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 automata controlling modes;
- `Function`, the functions of the dataflow; The functions can contains sub-functions. The topmost function (contained by the `FCMModel`) is considered as the main function.
- `FunctionPort`, the input and output 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 state machine guards.

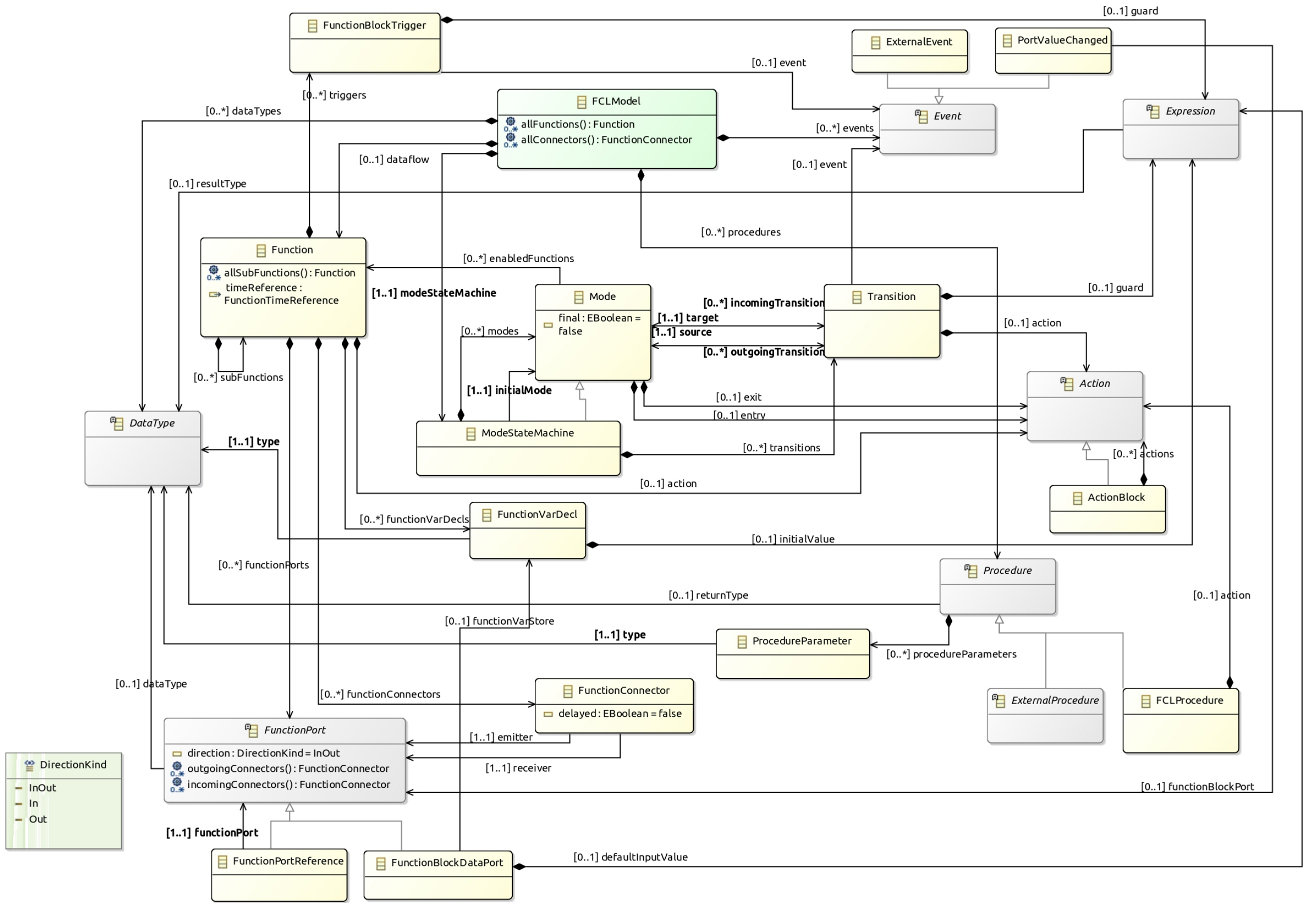

Figure 3. FCL main class diagram

Figure 4 shows the *ModeStateMachine* concepts. Basically, it models an automata controlling which Functions are enabled. 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*.

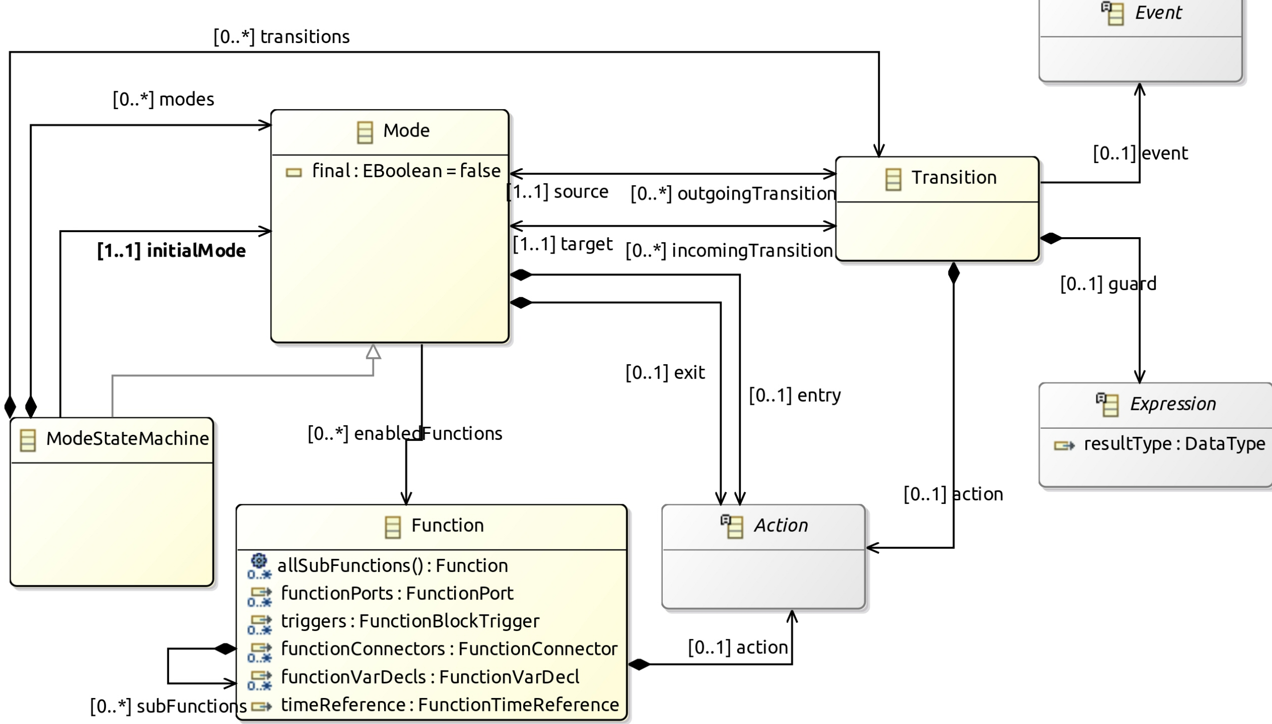

Figure 4. Mode automata class diagram

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.

A *Function* declares an *Action* whose main role is to compute the value to set on output ports based on the values of input ports.

Functions can declare a *Variable* that is local to the function. It allows to use it (read/write) from *Expressions* and *Action* (i.e. Event, procedure body, …). Modification of a variable is allowed to the current function or its children. A variable can also be modified by the dataflow if the variable is associated to a port. It can also be part of a *FunctionTimeReference*. In that case, the variable is automatically incremented when the function has been executed.

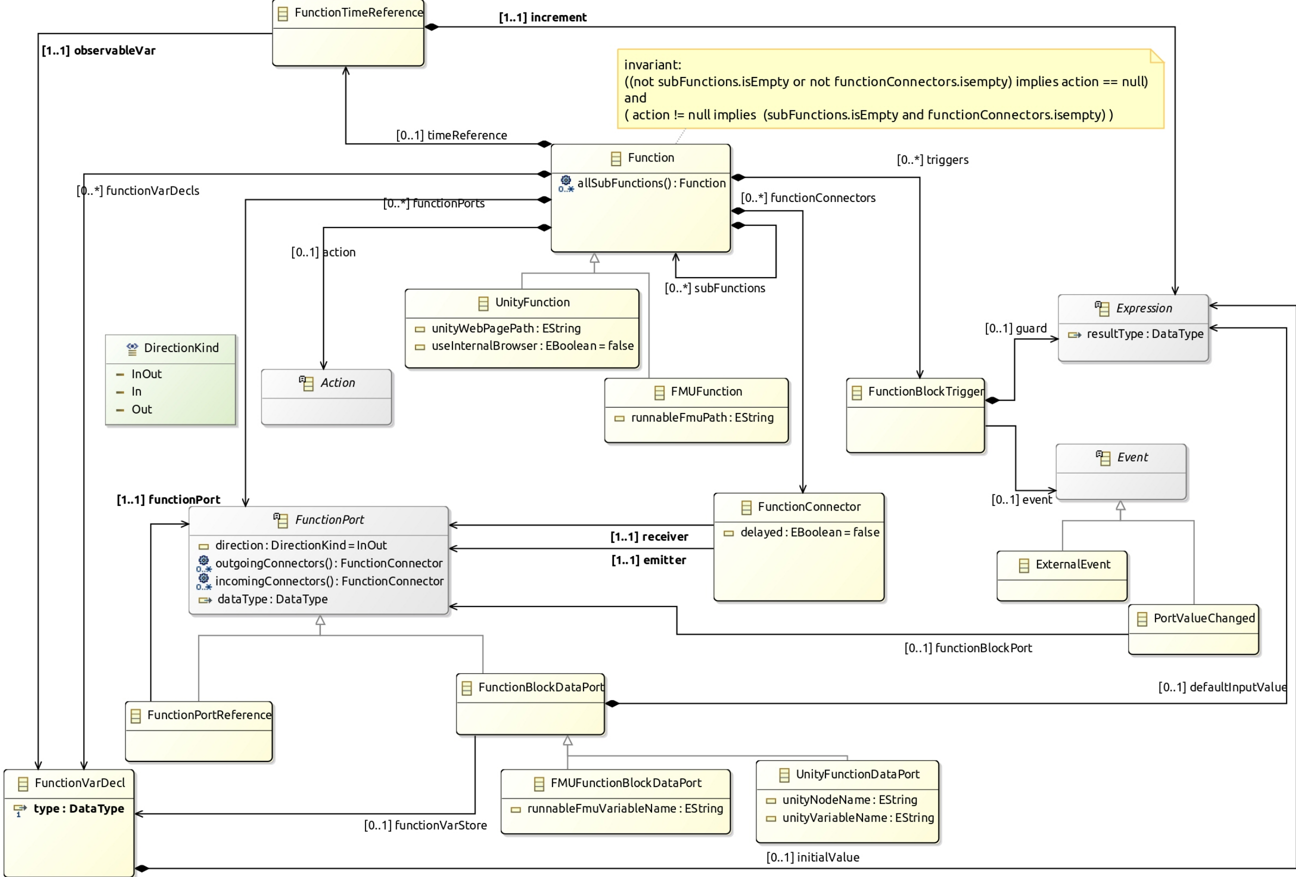

Figure 5. BlockFlow class diagram

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.

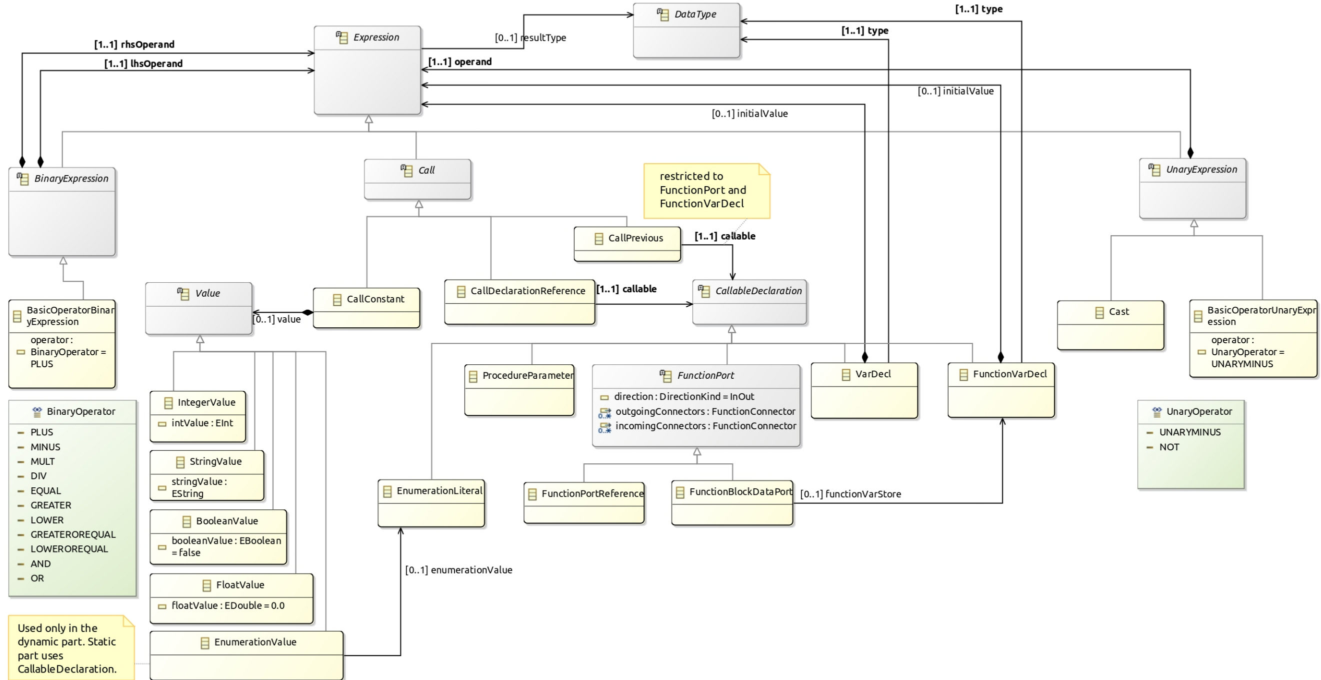

Figure 6. Expression class diagram

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.

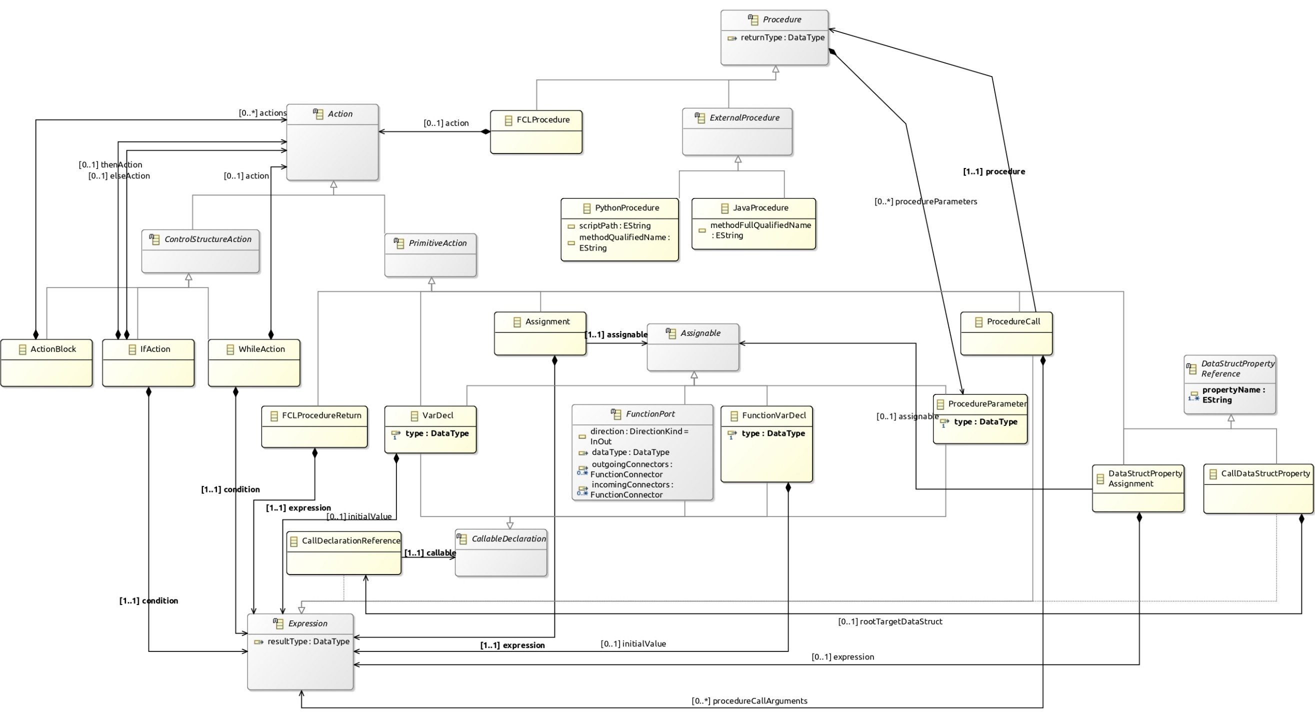

Figure 7. Action class diagram

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

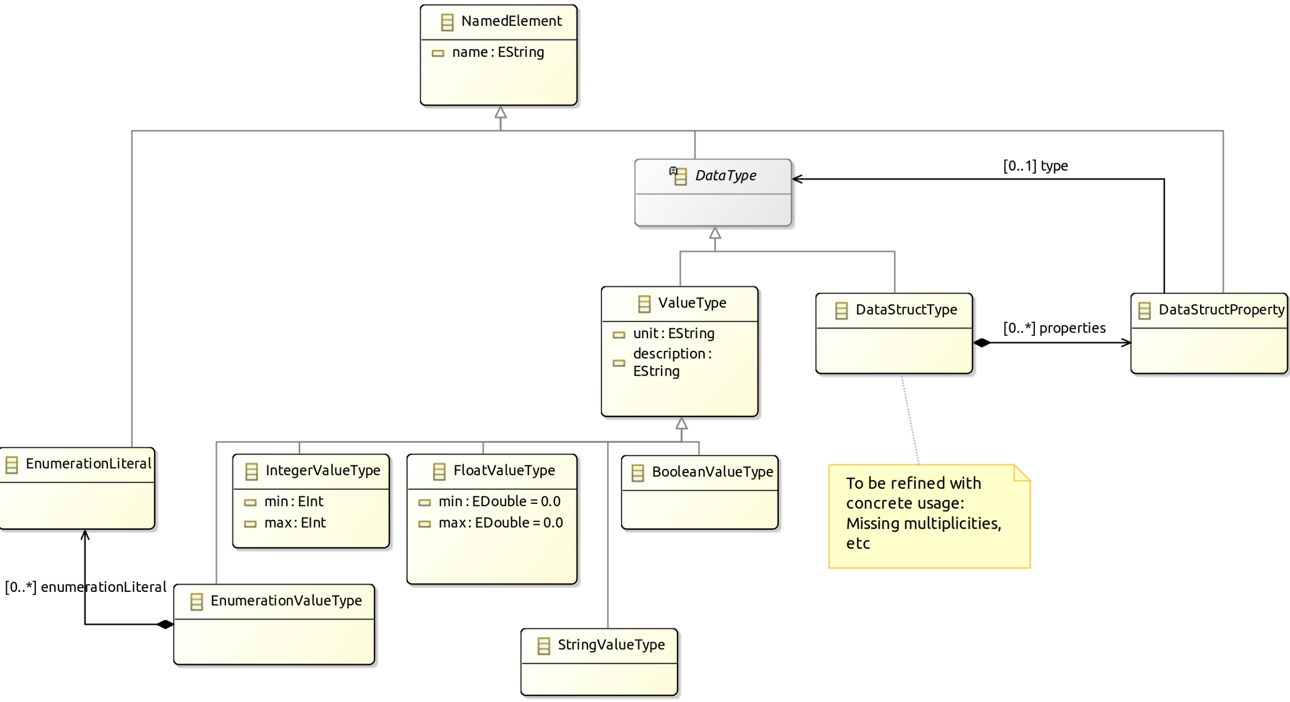

Figure 8. Types class diagram

2.1.2. Basic Tree Editor

Up to this point, the Ecore metamodel provides a minimalistic model edition support using a tree representation. Figure 9 shows this tree editor opened on the basic quadcopter controller (same model as in Figure 1).

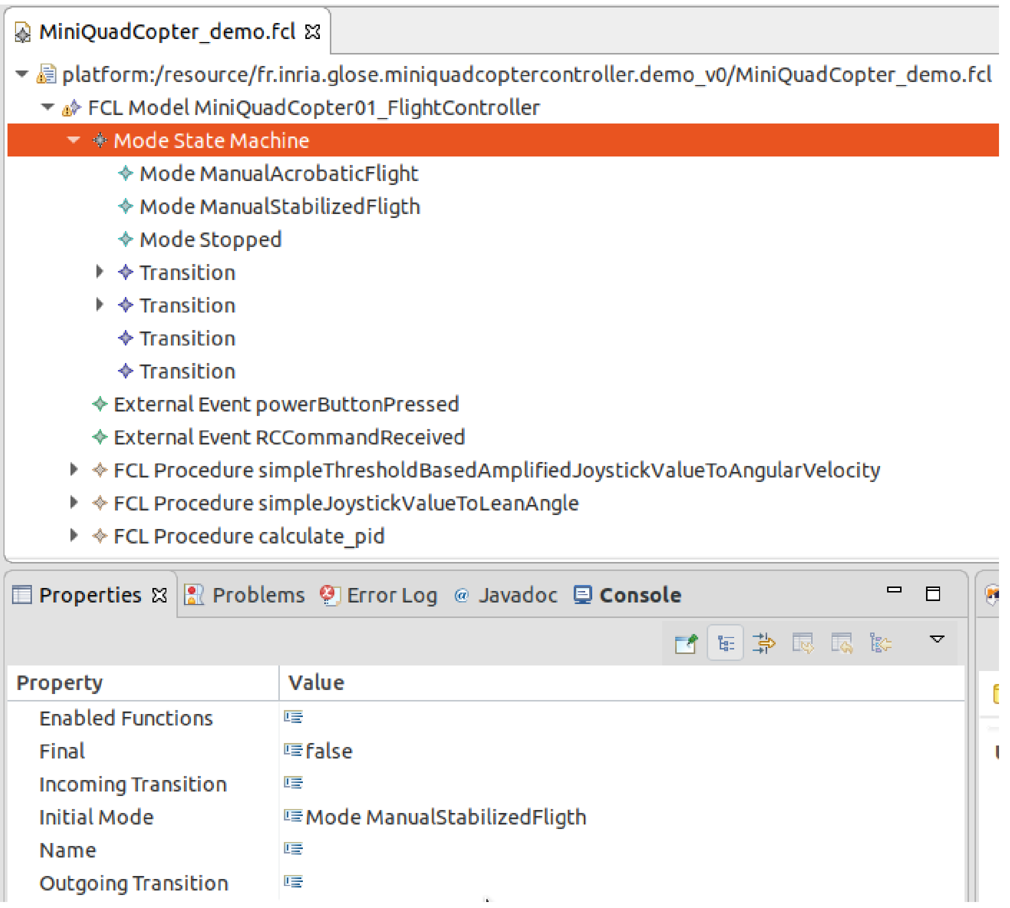

Figure 9. DemoController model in tree editor screenshot

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.

> Even if this demonstrator focuses on more sophisticated editors (Textual and graphical) (see Section 2.2), it is easy to customize the basic tree editor 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 `xxxItemProvider` 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. Editors Design

The demonstrator illustrates the development of two related and complementary editors:

- 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)

2.2.1. Xtext Editor

The demonstrator implements a textual editor for FCL using Xtext. Xtext is a framework based on a grammar language that produces an infrastructure with editing support for Eclipse. This infrastructure also offers interesting additional supports such as parser, linker, typechecker, or compiler support.

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 as they provide a syntax that is too verbose. They are customized to provide a friendlier textual syntax.

For example Listing XTtext rules for Action show some of the rules related to the Action part of the metamodel.

Listing XTtext rules for Action

```
Action returns Action:
    ActionBlock | Assignment | DataStructPropertyAssignment | IfAction | WhileAction |
ProcedureCall | VarDecl_Impl | FCLProcedureReturn | Log;

ActionBlock returns ActionBlock:
    {ActionBlock}
    '{'
        ( actions+=Action (  actions+=Action)*  )?
    '}';

Assignment returns Assignment:
    assignable=[Assignable|QualifiedName] ':=' expression=Expression
    ;
IfAction returns IfAction:
    'if' condition=Expression
    'then' thenAction=Action
    (=>'else' elseAction=Action )?
    ;
```

After having changed the Xtext rules, a Right click on the xtext file → *Run As* → *Generate Xtext artifacts* will update the java code for the parser.

Reference documentation for writing Xtext grammar rules: 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 format 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

The demonstrator implements a graphical editor for FCL using Sirius. Sirius is a framework based on a diagram specification language that provides graphical model edition support for Eclipse.

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 templates* → Select FCLModel as root container → *Finish*

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

Similarly to the default grammar produced by Xtext, this default diagram needs some improvements. The specifications for diagrams is located in the fcl.odesign file.

The improved odesign file of the demonstrator defines two diagrams:

- one diagram dedicated to show the Mode automata
- one diagram dedicated to show the DataFlow between the Function

Figure 10 shows the part of the odesign file specifying the automata. The property view shows how is configured the selected element. In the figure the TransitionEdge create an Edge linking the source attribute to the target attribute of *Transition*. This edge is represented using a *Solid* line. The ModeContainer uses several conditional style definition in order to customize the presentation according to various criterion (initial state, final state, …)

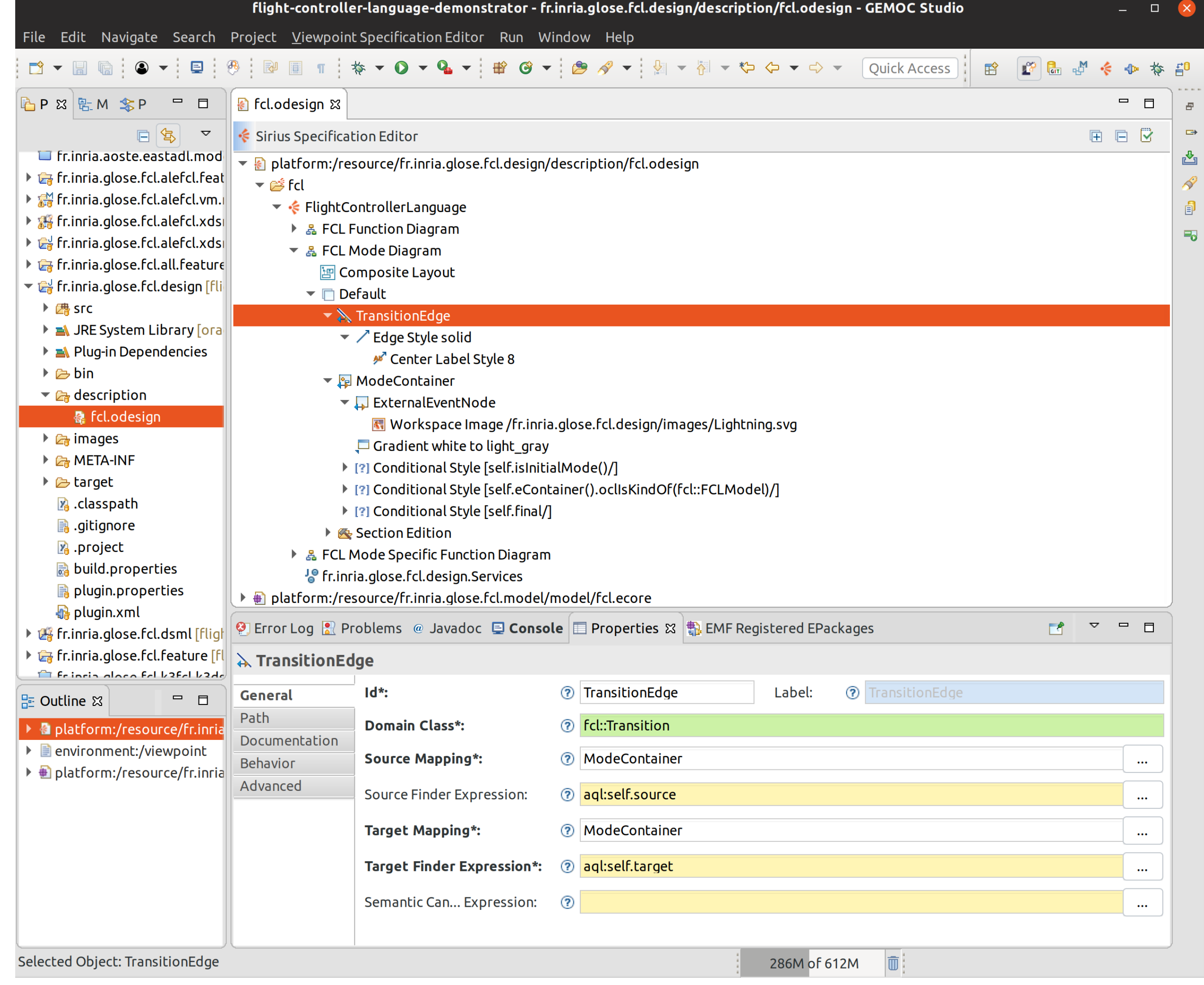

Figure 10. FCL Mode diagram specification screenshot

Figure 11 shows how the dataflow diagram is specified. It uses containers for the Functions as they can be nested. The ports are placed on the border of the container thanks to BorderedNode. Again, several conditional styles are used to differentiate the Function from FMUFunction. The dataflow diagram also defines several *Sections* that contains tools. A tool is actually an action that is added to the editor. For this diagram, the sections define creation actions allowing to create new element or connect them with the mouse.

Figure 11. FCL DataFlow diagram specification screenshot

Sirius uses queries in several places of the odesign file in order to compute various elements. For example to compute label strings or to navigate to a set of model elements. (for example in Figure 11, `self.eContents()` returns the list of all contained model elements of self) Writing queries directly in the fields of Sirius editor is sometime difficult to debug in case of complex query. For this reason, 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. They'll be easier to debug than internal queries as the classic java debugger will allow to inspect them in case of trouble.

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.

2.2.3. Sirius/Xtext Integration

The demonstrator implements several useful features in order to improve the collaboration of Xtext and Sirius editors: - use of Xtext serialization and formating in order to get the text to be used in Sirius (in labels or in popups). - navigation from the graphical editor to the textual editor. (ie. jump to the line in the textual editor corresponding to the selected graphical element)

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. This is done via a the definition of a tool in the diagram section specification (see Section 2.2.2).

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 doesn't try to modify the model 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. (see comments in the various diagrams of the metamodel)

Several techniques are possible (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 validation rules using the Xtext framework. They are in the file fr.inria.glose.fcl.xtext/src/fr/inria/glose/fcl/validation/FCLValidator.xtend. It contains validation rules that 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 (i.e. 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 validation rules 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.

2.3. Behavioral Semantic

The FCL language intends to be executable, i.e. supporting the execution of the conforming models. In the demonstrator, the semantics of the behavior is developed in ALE.

ALE is a language to complement Ecore metamodels with a behavioral semantics. Concretely, ALE allows to “re-open” the EClasses of a given Ecore metamodel to implement existing EOperations and weave new features or new operations.

The behavioral semantics is built as an extension of the Ecore metamodel defined in Section 2.1 thanks to the aspect mechanism of ALE.

Thanks to the behavioral semantics defined in ALE, it is straightforward to write an interpreter that implement a simulator for FCL models. However, the description of the complete development of a simulator based on the ALE behavioral semantics will be done in deliverable DP1.2: *Simulation and Compilation Environment for Flight Control Management* at T0+12.

The behavioral semantics is defined in the projects in fcl-implementations/alefcl

The main entry point is the fr.inria.glose.fcl.alefcl.xdsml/model/Alefcl.dsl file that defines the concerns of the language as shown in Listing Alefcl.dsl.

This file defines three kind of elements:

- a structural static part

- runtime data (i.e. dynamic information that can change during the model execution)
- operations (or rules) that manipulate the runtime data according to the static and dynamic parts.

Listing Alefcl.dsl

```
name = fr.inria.glose.fcl.alefcl.Alefcl                                        ①
ecore = platform:/resource/fr.inria.glose.fcl.model/model/fcl.ecore, \         ②
    platform:/resource/fr.inria.glose.fcl.alefcl.vm.model/model/alefcl_vm.ecore ③
ale=platform:/resource/fr.inria.glose.fcl.alefcl.xdsml/model/Alefcl.ale        ④
```

① Name of the language,

② Ecore metamodel of the static structure of the model,

③ Ecore metamodel that defines some of the runtime data using Ecore

④ a ALE file that defines:

- runtime data,
- operations that implement the expected behavior by manipulating the runtime data.

ALE is able to define all runtime data. However, in some situations, the runtime data can benefit from being placed in a separate Ecore. (For example, to present a complex runtime data structure as a diagram) The file `alefcl_vm.ecore` illustrates this.

Listing Alefcl.ale entrypoint shows the starting point of the behavior defined in fr.inria.glose.fcl.alefcl.xdsml/model/Alefcl.ale.

Listing Alefcl.ale entrypoint

```
behavior alefcl;

//Import external Java services
use fr.inria.glose.fcl.alefcl.xdsml.services.FCLService;
use fr.inria.glose.fcl.alefcl.xdsml.fmu.services.FMUService;

open class fcl.FCLModel {                                   ①

    1 ..* alefcl::EventOccurrence receivedEvents;           ②
    0 ..* fcl::ProcedureContext procedureStack;

    Integer elapsedTime;
    Boolean sigTermReceived;

    String indentation; // used to improve console display

    @main
    def void main() {                                       ③
        self.devInfo('-> main() '+self.name);
```

```
        // while not on a final mode
        while(not self.modeStateMachine.currentMode.final){
            self.doMainLoop();
            // no one will update currentValues anymore,
            self.dataflow.updatePrevValues();
        }
    }

    @step
    def void doMainLoop(){                                        ④
        self.info('##################################################');

        Sequence(fcl::Transition) fireableModeTransitions :=
            self.modeStateMachine.fireableModeTransitions().transitions;

        while(fireableModeTransitions->size() >0 or self.receivedEvents->size() > 0){
            self.devInfo(self.indentation+
            '\t fireableTransitions='+fireableModeTransitions+ ' (out of
'+self.modeStateMachine.currentMode.outgoingTransition+')');

            if(fireableModeTransitions->size() >0){
                self.modeStateMachine.switchMode(fireableModeTransitions->first());
                // re-evaluate possible mode switch
                fireableModeTransitions :=
self.modeStateMachine.fireableModeTransitions().transitions;
            }
            while(fireableModeTransitions->size() = 0 and self.receivedEvents->size()
> 0){
                // flush events that cannot be directly used to switch mode
                self.devInfo('no more transition can be fired, but some event remains:
consume one');
                self.consumeEventOccurence();
                fireableModeTransitions :=
self.modeStateMachine.fireableModeTransitions().transitions;
            }
        }

        self.devInfo('no more mode switch are possible and all events have been
consumed');
        self.devInfo('current mode is '+self.modeStateMachine.currentMode.name);
        self.devInfo('with the following enabled
functions:\n\t'+self.modeStateMachine.currentMode.enabledFunctions-
>collect(f|f.getQualifiedName())->sep('\n\t'));
        self.info('### Evaluate Main dataflow '+self.dataflow);

        self.dataflow.evalFunction();                                   ⑤

        self.dataflow.sendDataThroughAllDelayedConnectors();            ⑥


    }
```

```
    @init
    def void initializeModel(String input) {                          ⑦
        self.modeStateMachine.currentMode := self.modeStateMachine.initialMode;

        // init default values of functionVars and connectors
        self.dataflow.initFunctionVars();
        self.dataflow.initPortsDefaultValues();

        self.dataflow.updatePrevValues();

        // init FMUs
        for(fmuFunction in self.dataflow.eAllContents(fcl::FMUFunction)){
            fmuFunction.initializeFMURuntime();
        }

        // TODO optional load of a kind of scenario of input events...
    }
```

① reopen class `FCLModel` defined in `fcl.ecore` in order to add new attributes and operations

② new attributes that will be part of the runtime data: a list of EventOccurence received by the system and a contextual stack used by the behavior of `fcl.Procedure`.

③ the main rule that is launched when executing a model. It basically does a loop of the until the state machine reaches a final state. On each iteration, it calls the `doMainLoop()` operation and manages the storage of previous values (required to implement the `prev` function supported by FCL)

④ an operation gathering the main loop behavior. This operation is annotated with `@step`, which means that if the execution is run in debug mode, this operation is observable. It can pause or be used as part of a breakpoint.

⑤ call to the evaluation of the main function.

⑥ all delayed connections are processed once all the functions have been evaluated

⑦ this method is launched before starting the execution. Its role is to initialize the runtime data according to the model default values. Optionally, launch arguments can be used to parameterize this model initialization.

The operation `doMainLoop()` firstly evaluates all the possible mode switches. A mode switch may consume event, if after having processed all the possible mode switches some EventOccurence remains, they are flushed. The operation then evaluates the main Function, i.e. it computes the data flow. As shown in Listing Alefcl.ale fcl.Function.evalFunction() this main Function processes its internal functions recursively after having sorted them using a topological sort.

Listing Alefcl.ale fcl.Function.evalFunction()

```
open class fcl.Function {

    @step
    def void evalFunction(){                                              ①
        self.info('evaluating Function '+self.functionName()+'...');

        if(self.timeReference <> null){                                   ②
            self.devInfo('Increment time reference for function
'+self.functionName());
            fcl::FunctionVarDecl timeVariable := self.timeReference.observableVar;
            timeVariable.assign(
                    timeVariable.currentValue.plus(
                        self.timeReference.increment.evaluate()
                    )
            );
        }

        if(self.subFunctions->size() > 0){
            self.sendInputDataToActiveInternalFunctions();               ③

            // runtocompletion of nested functions
            // topological sort of functions
            alefcl_vm::FunctionSequence sortedF := self.sortedSubfunctions();   ④
            self.info('   will evaluate functions in the following order:
'+sortedF.functions->collect(f|'"'+f.name+'"')->sep(', '));


            for(f in sortedF.functions){
                f.evalFunction();                                         ⑤
            }
        }

        if(self.action != null) {
            fcl::ProcedureContext procedureContext := fcl::ProcedureContext.create();
            // push context
            self.eContainer(fcl::FCLModel).procedureStack +=  procedureContext;
            // run the action in this context
            self.action.doAction();                                       ⑥
            // pop context
            self.eContainer(fcl::FCLModel).procedureStack -=  procedureContext;
        }

        // for all non delayed outgoing connections, send data
        self.sendDataToActiveNonDelayedPorts();                           ⑦

    }
```

① `evalFunction()` operation is added to `fcl.Function` defined in `fcl.ecore`

② a function may declare a clock, it is incremented here

③ send input data to the ports of active internal sub functions.

④ make sure to run internal sub function in the correct order using a topological sort. this takes care to ignore functions that are not enabled for the current mode.

⑤ concrete evaluation of sub functions

⑥ evaluate the action associated to the current function. This is done in context stack.

⑦ send the data to the active ports of function connected to this one. Ignore delayed connections that are processed in `fcl.FCLModel.doMainLoop()` see Listing Alefcl.ale entrypoint)

Listing Alefcl.ale current and previous values shows that current values and previous value are added as runtime data directly on the relevant concepts of `fcl.ecore`: `FunctionBlockDataPort` and `FunctionVarDecl`.

Listing Alefcl.ale current and previous values

```
open class fcl.FunctionBlockDataPort extends fcl.FunctionPort{
    fcl::Value previousValue;
    fcl::Value currentValue;
[...]

open class fcl.FunctionVarDecl {
    fcl::Value previousValue;
    fcl::Value currentValue;
[...]
```

Listing Alefcl.ale binary expression evaluation shows the principle of Expression evaluation. `BasicOperatorBinaryExpression.evaluate()` implements the operators by delegating to the corresponding operation in Expression. Most of the operations in Expression are logically abstract (i.e. the body actually raises an exception inviting the Language engineer to implement the possibly missing operation) as they are actually implemented on each sub class (IntegerValue, BooleanValue, etc)

Listing Alefcl.ale binary expression evaluation

```
open class fcl.BasicOperatorBinaryExpression extends fcl.BinaryExpression {
    //override DataValue evaluate() {
    def fcl::Value evaluate() {
        self.devInfo(self.eContainer(fcl::FCLModel).pushIndent()+
            '-> BasicOperatorBinaryExpression.evaluate '+self);
        if(self.operator = fcl::BinaryOperator::OR) {
            fcl::BooleanValue bValue := fcl::BooleanValue.create();
            bValue.booleanValue := self.lhsOperand.evaluateAsBoolean() or
self.rhsOperand.evaluateAsBoolean();
            result := bValue;
        } else if(self.operator = fcl::BinaryOperator::AND){

            fcl::BooleanValue bValue := fcl::BooleanValue.create();
            bValue.booleanValue := self.lhsOperand.evaluateAsBoolean() and
```

```
self.rhsOperand.evaluateAsBoolean();
            result := bValue;
        } else {
            // all other operators need both lhs and rhs evaluation
            fcl::Value lhs := self.lhsOperand.evaluate();
            fcl::Value rhs := self.rhsOperand.evaluate();
            if(self.operator = fcl::BinaryOperator::EQUAL){
                result := lhs.equals(rhs);
            } else if(self.operator = fcl::BinaryOperator::PLUS){
                result := lhs.plus(rhs);
            } else if(self.operator = fcl::BinaryOperator::MINUS){
                result := lhs.minus(rhs);
            } else if(self.operator = fcl::BinaryOperator::MULT){
                result := lhs.mult(rhs);
            } else if(self.operator = fcl::BinaryOperator::DIV){
                result := lhs.div(rhs);
            } else if(self.operator = fcl::BinaryOperator::GREATER){
                result := lhs.greater(rhs);
            } else if(self.operator = fcl::BinaryOperator::LOWER){
                result := lhs.lower(rhs);
            } else if(self.operator = fcl::BinaryOperator::GREATEROREQUAL){
                result := lhs.greaterOrEquals(rhs);
            } else if(self.operator = fcl::BinaryOperator::LOWEROREQUAL){
                result := lhs.lowerOrEquals(rhs);
            } else {
                self.raiseException('not implemented, please implement evaluate() for
'+self);
            }
        }
        self.eContainer(fcl::FCLModel).popIndent();

    }
}
open class fcl.Value {
    //override DataValue evaluate() {
    def fcl::Value evaluate() {
        self.devError('not implemented, please ask language designer to implement
evaluate() for '+self);
        self.raiseException('not implemented, please implement evaluate() for '+self);
    }
[...]

    def fcl::Value plus(fcl::Value rhs) {
        self.devError('not implemented, please ask language designer to implement
plus() for '+self);
        self.raiseException('not implemented, please implement plus() for '+self);
    }
    def fcl::Value minus(fcl::Value rhs) {
        self.devError('not implemented, please ask language designer to implement
minus() for '+self);
        self.raiseException('not implemented, please implement minus() for '+self);
```

```
    }
    def fcl::Value mult(fcl::Value rhs) {
        self.devError('not implemented, please ask language designer to implement
mult() for '+self);
        self.raiseException('not implemented, please implement mult() for '+self);
    }
[...]
}
open class fcl.IntegerValue {
    def fcl::Value evaluate() {
        self.devInfo(self.eContainer(fcl::FCLModel).indentation+
            ' -> IntegerValue.evaluate '+self);
        result := self;
    }
    def fcl::Value copy() {
        fcl::IntegerValue aValue := fcl::IntegerValue.create();
        aValue.intValue := self.intValue;
        result := aValue;
    }
    def fcl::BooleanValue equals(fcl::Value rhs) {
        fcl::BooleanValue bValue := fcl::BooleanValue.create();
        bValue.booleanValue := self.intValue = rhs.toIntegerValue().intValue;
        result := bValue;
    }
    def fcl::BooleanValue lower(fcl::Value rhs) {
        fcl::BooleanValue bValue := fcl::BooleanValue.create();
        bValue.booleanValue := self.intValue < rhs.toIntegerValue().intValue;
        result := bValue;
    }
[...]
```

As an overview of DP1.2, running this operational semantics using a GEMOC based engine will produce executions similar to Figure 12.

Where `self.devInfo()`, `self.info()` and `self.important()` messages are logged in the Console View in green, black and blue respectively. Other views are used to interact with the execution or display values.

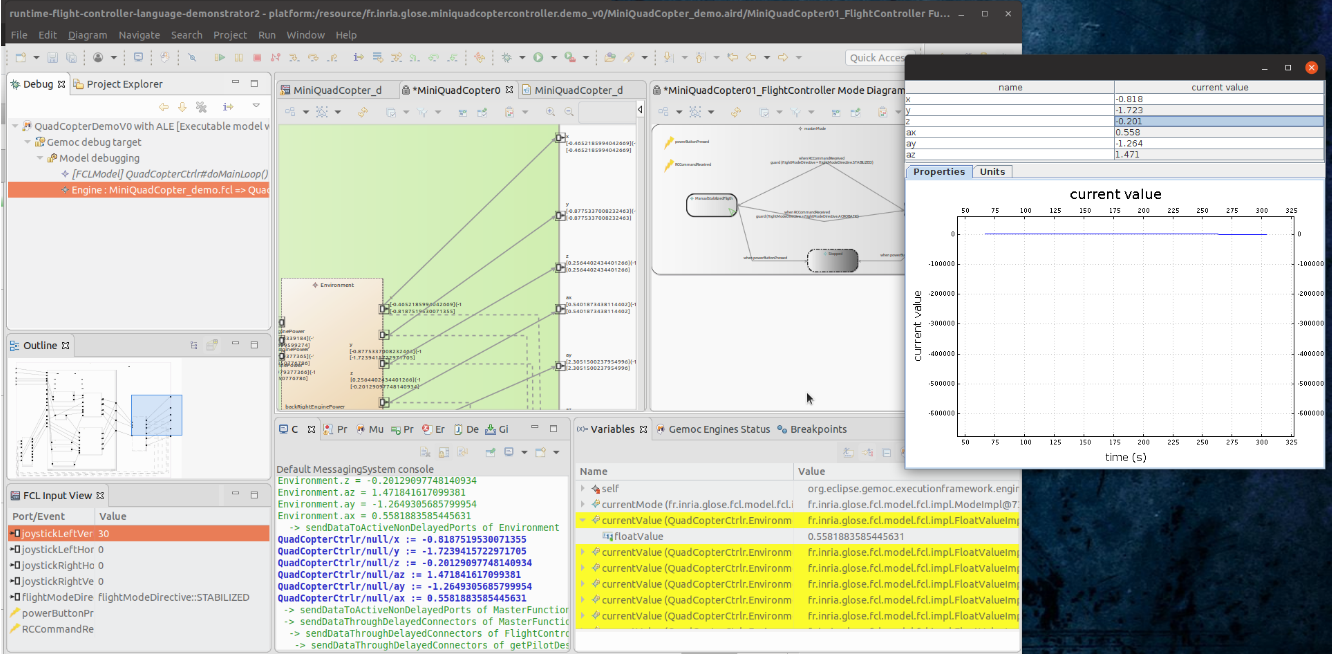

Figure 12. Demo controller model running and paused after having done one `doMainLoop()`.

2.4. Deployment

The modeling environment can be deployed in two ways:

- for development purpose, a runtime workbench is started from the language workbench. It allows to test and debug the language modeling environment.
- for use by the final end user (i.e. the Modeler), the projects can be packaged and installed directly in an Eclipse product (a.k.a. RCP or Rich Client Platform).

2.4.1. Packaging and Deployment in an Eclipse Product

In order to package the FCL language for final user, one of the easiest way is to prepare an update site and upload it on a web server. Then, the user will be able to install the FCL modeling environment in any Eclipse IDE. As the FCL language is based on EMF, Sirius, Xtext and GEMOC, the runtime of these framework must be installed too. If not present in the choosen IDE, they can be installed using updatesite too.

The GEMOC Studio (http://gemoc.org/download.html) is the preferred IDE as it already contains all the prerequisite frameworks.

In the demonstrator, building this updatesite is done via the maven script of the releng/fr.inria.glose.fcl.updatesite project.

The resulting *repository* folder can then be communicated to the user (typically via a web URL) so he can use the *Help → Install New Software…* menu of his Eclipse IDE as in Figure 13.

Figure 13. Install features from update site

In Figure 13, type the following URL as update site location to get the latest version of the FCL language: https://glose.gitlabpages.inria.fr/flight-controller-language-demonstrator/updatesite/latest/

The update site can also be advertised via the Eclipse Market Place (https://marketplace.eclipse.org/). This is a central place where developers can publish their contribution to the Eclipse ecosystem that is very convenient for end users.

Thanks to Eclipse RCP (Rich client Platform) technology, it is possible to create an IDE containing only the tools required for an activity. For example, as the Modeler can have an IDE with only the runtime and not the developer version for Java, Xtext, Sirius,...

2.4.2. Runtime Deployment

If you have imported all the source projects of the FCL language into an Eclipse IDE, to deploy it in a runtime workbench the Language engineer must do:

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

This creates a launch configuration (see [new-runtime-workbench-launchconf]) that runs a new Eclipse similar to the current one but also that also includes the plugins under development in the workspace.

RuntimeWorkbench-RunConfiguration-Screenshot.png

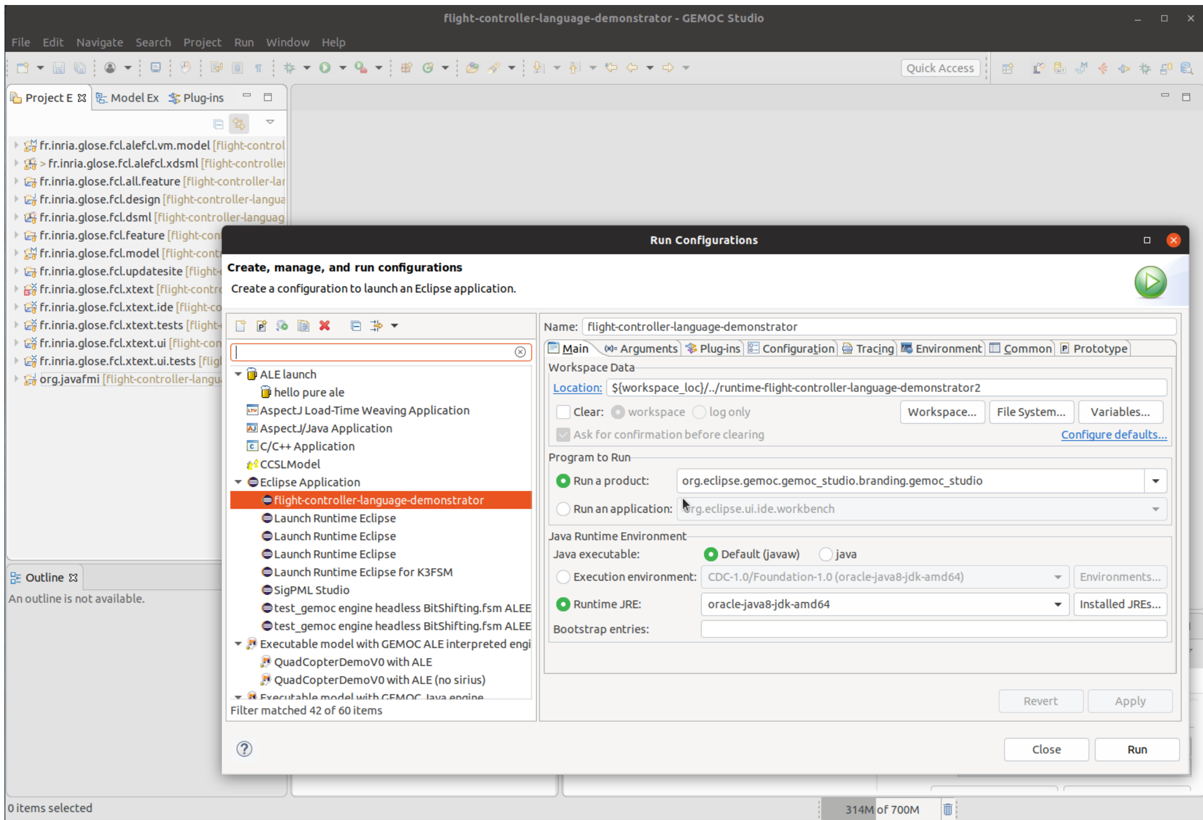

Figure 14. Create new runtime configuration for a modeling workbench

Chapter 3. Scenario #2: Editing Model

This scenario demonstrates the ability of the GEMOC Studio to provide a full-fledged modeling environment, that supports a language user (i.e., a domain expert, being a system engineer in the case of the chosen use case) in the design of a complex system.

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 slightly more complex example would look like Listing Small FCL model with one main function. It defines

Listing Small FCL model with one main 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 main 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 two sub functions (in this case they are exclusive).

⑤ each sub function declares 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 two children functions).

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

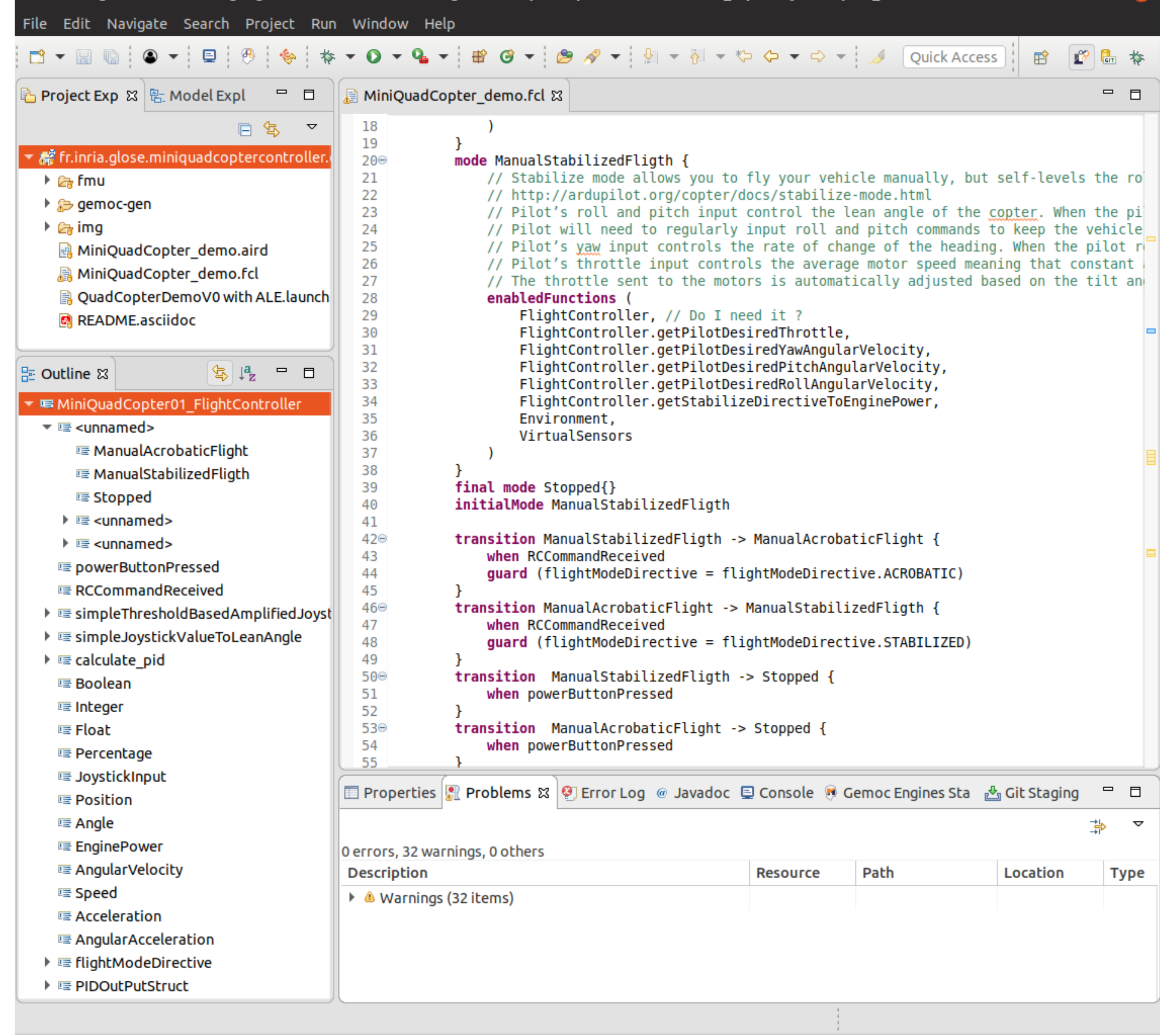

Figure 15. Xtext editor opened on the demo controller (mode state machine part)

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 16. A popup menu on every element or using the palette *Open in textual editor* allows to highlight the text of element in Xtext. The palette of the Function diagram also offers to create new elements (New Functions or new Ports) and to create a connection between ports.

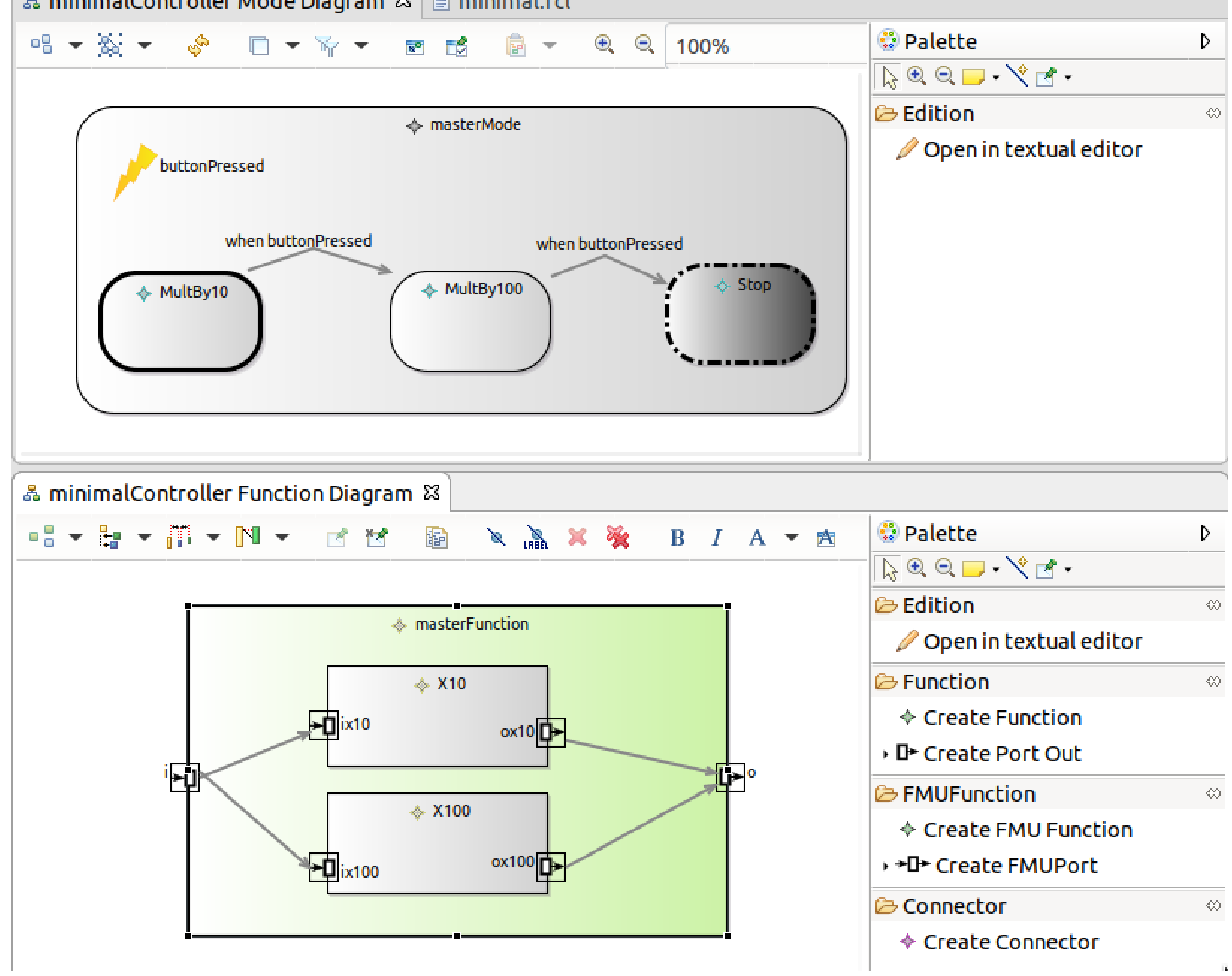

Figure 16. Small FCL model opened with the graphical editor.

More complex diagram would look like Figure 1 and Figure 2 which are overview of the models that will be released as part of the Example Flight Control Language models of DP3.1. They show the graphical editor diagrams applied to a Quadcopter controller.

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*

Figure 17 shows the type checker indicating a type error while trying to assign a Float in an Integer variable of FCL (the type *JoystickInput* is an Integer). Using a *Right click* → *Validate diagram* in the Sirius editor will also highlight the problem in the graphical editor. Each markers of both editor (textual and graphical) is reported in the *Problems view*.

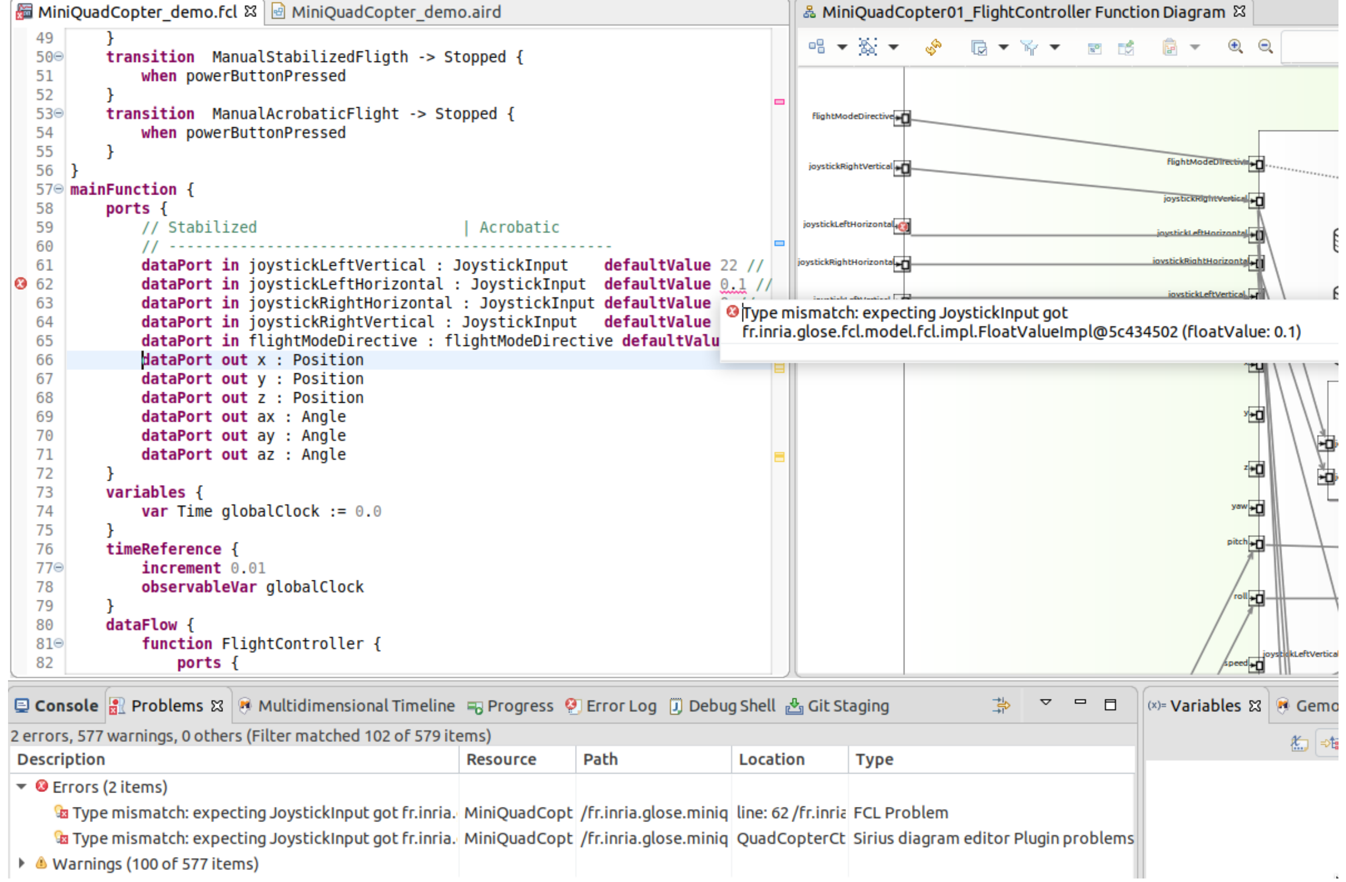

Figure 17. Type mismatch error in the editors and Problems view

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 18 shows this tree editor on the basic quadcopter controller.

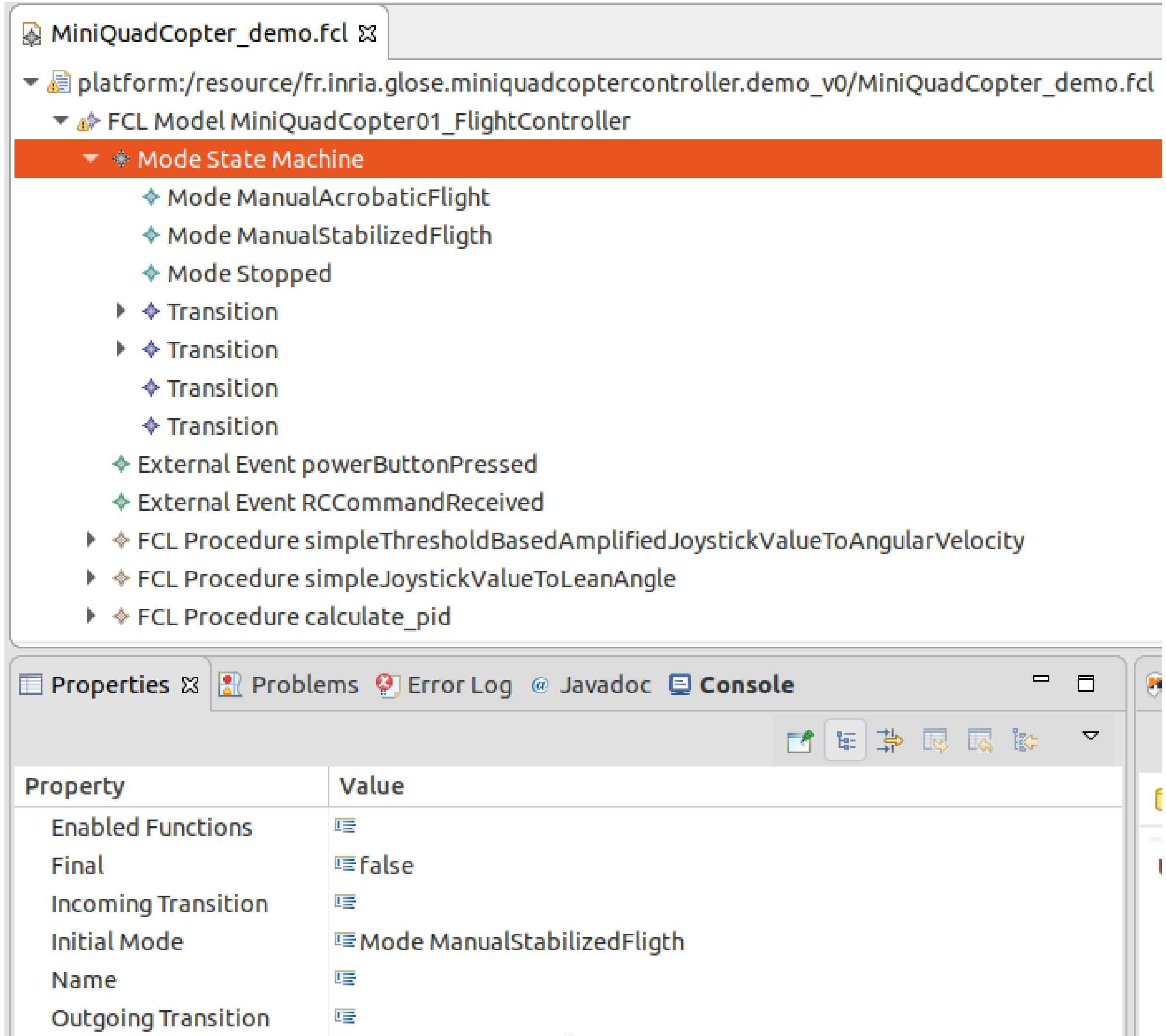

Figure 18. DemoController model in tree editor screenshot

Chapter 4. Additional features scheduled for T0+12

The global process of the demonstrator is iterative. This means that the other deliverables will contribute to the process described here. As part of the V2 of the demonstrator at T0+12, a new version of this document will be released in order to integrate them.

Progress of most of the improvements are listed in the gitlab: https://gitlab.inria.fr/glose/flight-controller-language-demonstrator/issues

4.1. Enhancement of Existing Features

Improved Mode diagram
: The existing mode diagram would benefit from having several enhacements. For example, similarly to the dataflow diagram, it could provide a tool palette for element creation.

Improved DataFlow diagram
: The existing dataflow diagram would benefit from having several enhacements. For example, it could provide a way to identify the functions enabled by a given mode.

4.2. New Features

Scenario Simulating and Debugging a Behavioral Model

This scenario is mainly developed for deliverable DP1.2. It will also introduce several concerns that collaborates with the element described in the current document. For example, in GEMOC, the debug and animation visualizations are relatively easy to develop as Sirius diagram extensions. These extensions will provide new layers on top of the graphical editor in order to display runtime and debug information.

Scenario Designing concurrent executable languages

This scenario is mainly developed for deliverable DP2.1. The support of concurrency will require some changes in the specification of the behavioral semantics defined in Section 2.3.

Scenario Exploring the Design Space of a Behavioral Model

This scenario is developed for deliverable DP3.1 (with strong connection to DP1.2 and DP2.1). It will require to iterate on the specification of the behavioral semantics defined in Section 2.3. This scenario will demonstrate the ability of the GEMOC Studio to explore the possible execution paths allowed by the model.The demonstrator will show how to define the event interface in the language workbench and then how to record external stimuli as generic scenarios and reuse them as test cases in the modeling workbench.