Grant ANR-12-INSE-0011

ANR INS GEMOC

D4.2.1 - Generic Engine for Heterogeneous Models Execution (SOFTWARE)

Task 4.2.1

Version 1.0

DOCUMENT CONTROL

	–: 2013/11/28	A:	B: 2014/10/31	C:	D:
Written by Signature	Florent Latombe	Florent Latombe	Didier Vojtisek		
Approved by Signature					

Revision index	Modifications
–	version 0.1 — Initial version
A	version 0.2 — Reworked most chapters
B	version 1.0 — Reworked all chapters according to current implementation
C	
D	

Authors

Author	Partner	Role
Didier Vojtisek	Inria	Lead author
Florent Latombe	IRIT - Université de Toulouse	Contributor
Xavier Crégut	IRIT - Université de Toulouse	Contributor
Marc Pantel	IRIT - Université de Toulouse	Contributor

Contents

1 Introduction 5
1.1 Purpose 5
1.2 Definitions, Acronyms and Abbreviations 5
1.3 Reminders 6
1.3.1 xDSML Definition 6
1.3.2 Available Implementations 6

2 General architecture 8
2.1 Components overview 8
2.2 MSE Engine 9
2.3 Execution Platform 9
2.4 Policy / Decider 9
2.4.1 Random decider 9
2.4.2 Interactive decider 9
2.4.3 Solver decider 9
2.5 MoCC/DSE compiler 9
2.6 Solver 9
2.7 Composition Operator applicator 9

3 MSE Engine details 10
3.1 Service view 10
3.1.1 Mandatory services 10
3.1.2 Optionnal services 11
3.1.3 Debug 11
3.1.4 Dependency inversion 12
3.2 Execution flow 12
3.2.1 Initialization 12
3.2.2 One global step of execution 12
3.2.3 Observe engine behavior and notification 13

4 Execution configuration 14
4.1 Configuring the engine from the Language Workbench 14
4.1.1 Configuring the engine with the basic xDSML editor 14
4.1.2 Configuring the engine with the advanced xDSML editor 15
4.1.3 Adding hooks to the engine 15
4.2 Configuring the engine from the Modeling Workbench 15

5 Implementation and distribution 18

6 Perspectives and issues 21
6.1 Replay Decider 21
6.2 IModelLoder 21
6.3 Debug 21

7 Conclusion 22

1. Introduction

The GEMOC Execution Engine is a generic platform which aims at being able to execute any model conforming to any eXecutable Domain Specific Modeling Language (xDSML) defined using the GEMOC language workbench (see Deliverable 1.2.1 for detailed informations about the tools) and the corresponding methodology (see Deliverable 1.1.1 for detailed informations about the definition of an xDSML). In particular, its goals include being able to execute a composition of such well-defined xDSMLs.

1.1 Purpose

This document describes the v1 of the software deliverable D4.2.1 (*Generic Engine for Heterogeneous Models of Execution*). It presents the current state of the Execution Engine in the GEMOC Studio.

As it requires several other components to run, the engine is distributed as part of a full GEMOC Studio. It is available for download from http://gemoc.org/studio/studio-download/

The present document corresponds to the version V0.1.2 2014_11_14.

1.2 Definitions, Acronyms and Abbreviations

- **AS**: Abstract Syntax.
- **API**: Application Programming Interface.
- **Behavioral Semantics**: see *Execution semantics*.
- **CCSL**: Clock-Constraint Specification Language.
- **CS**: Concrete Syntax.
- **Domain Engineer**: user of the Modeling Workbench.
- **DSA**: Domain-Specific Action.
- **DSE**: Domain-Specific Event.
- **DSL**: Domain-Specific Language.
- **DSML**: Domain-Specific (Modeling) Language.
- **Dynamic Semantics**: see *Execution semantics*.
- **Eclipse Plugin**: an Eclipse plugin is a Java project with associated metadata that can be bundled and deployed as a contribution to an Eclipse-based IDE.
- **ED**: Execution Data (part of DSA).
- **EF**: Execution Function (part of DSA).
- **Execution Semantics**: Defines when and how elements of a language will produce a model behavior.
- **GEMOC Studio**: Eclipse-based studio integrating both a language workbench and the corresponding modeling workbenches.
- **GUI**: Graphical User Interface.

- **Language Workbench**: a language workbench offers the facilities for designing and implementing modeling languages.
- **Language Designer**: a language designer is the user of the language workbench.
- **MoCC**: Model of Concurrency and Communication.
- **Model**: model which contributes to the convent of a View.
- **Modeling Workbench**: a modeling workbench offers all the required facilities for editing and animating domain specific models according to a given modeling language.
- **MSA**: Model-Specific Action.
- **MSE**: Model-Specific Event.
- **RTD**: RunTime Data.
- **Static semantics**: Constraints on a model that cannot be expressed in the metamodel. For example, static semantics can be expressed as OCL invariants.
- **Operational semantics**: Constraints on a model that defines its step-by-step execution semantics (see **Execution Semantics**).
- **TESL**: Tagged Events Specification Language.
- **xDSML**: Executable Domain-Specific Modeling Language.
- **MoCCML**: MoCC Modeling Language.

1.3 Reminders

1.3.1 xDSML Definition

Detailing the definition of an xDSML is not within the scope of this document. Please refer to Task 1.1.1 for a finer-grained description.

An xDSML is made of an Abstract Syntax (AS), Domain-Specific Actions (including execution functions and data) as well as Domain-Specific Events (DSEs) mapping DSAs with well-defined Models of Concurrency and Communication (MoCCs) (*cf.* WP2), and a Feedback Model which specifies the influence of the return values from the Domain-Specific Actions on the future of the execution.

1.3.2 Available Implementations

However, all the different technologies which can be used to implement the above-mentioned language units were not all initially designed with the intent to interact with each other. One of the technical difficulties in the GEMOC Execution Engine is to be able to execute any model conforming to an xDSML designed using the GEMOC softwares and technologies, without regards to the precise technologies used for implementation of each component. In order to take these concerns into account, the engine defines several APIs that the components need to implement in order to interact with the engine.

In this version, we have limited our tests to a limited set of technologies. We focus on the technologies we know best in order to validate our approach. However, the engine APIs are flexible enough to let the user define his own implementation that'll support other technologies with minimal effort. We have tested the current version of the engine with several technologies. For all of them, the engine comes with the appropriate APIs implementations. In addition, the default technologies proposed by the GEMOC process (see Deliverable 1.1.1) have some dedicated tooling that automate the development.

Here are the technologies we have tested with the current version of the engine:

1.3.2.1 MoC Implementations

- The MoCCML Language as defined in Deliverable 2.2.1.
- The CCSL language that is internally used by TimeSquare.

1.3.2.2 DSAs Implementations

- Kermeta 3
- Java with the EMF API

1.3.2.3 DSE Implementations

- At first we only support ECL.
- We plan to replace ECL with a more generic and tailored-to-our-needs language (cf. Deliverable 1.3.1)

2. General architecture

2.1 Components overview

As outlined by the GEMOC Studio architecture Figure 2.1 (from Deliverable 4.1.1 V1.1), the Execution Engine aggregates a set of components in order to offer an execution platform. It also need the support from the Modeling Worbench components since the execution take place in this workbench and is displayed by the provided editors and animators.

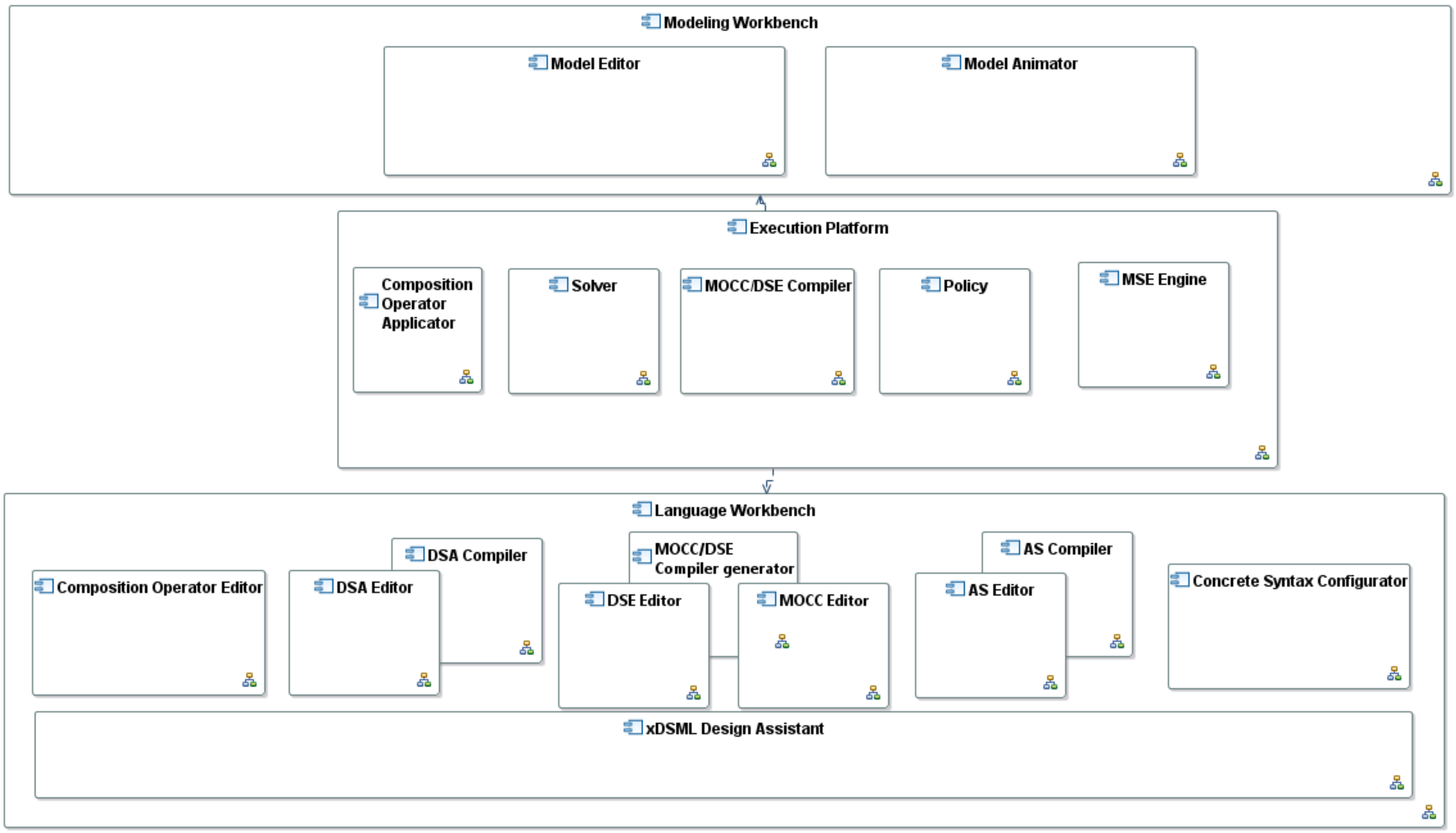


Figure 2.1: Architecture of the GEMOC Studio (from D4.1.1 - V1.1)

We remind in the following sections the main purpose of the components that are directly involved in the Generic Engine for Heterogenous Models Execution. The relationship between all the components of the architecture are described in Deliverable 4.1.1,

2.2 MSE Engine

The `MSE Engine` acts on an `Execution Model` and runs the appropriates DSAs according to the MSEs scheduled by the solver.

It can produce an execution trace. In order to be generic, many part of the DSE Engine are delegated to specific services in the other components.

As this component orchestrates the use of the other ones, it currently plays a central role in the Execution Platform. The technical details of this component will be presented in Chapter 3.

2.3 Execution Platform

This is an abstract component that represents the assembly of the components required to offer an execution platform. Its only specific role is to provide a GUI for configuring the components for a given execution. This configuration activity will be detailed in Chapter 4.

As this component is the main entry for all execution activities, it may be called *Execution Engine* in order to refer to the global set of components without looking precisely to a given component.

2.4 Policy / Decider

This component offers several strategies to decide which LogicalStep must be executed when, at a given execution step, the solver proposes more than one possible LogicalStep for execution.

A LogicalStep is a set of MSE Occurences, ie. a set of simultaneous MSEs.

Several selection strategies are provided in GEMOC :

2.4.1 Random decider

This simple strategy will select randomly one LogicalStep. The drawback of such Policy is that it may not be reproductible however it may be interesting for stress test of the model.

2.4.2 Interactive decider

The purpose of the Interactive Policy is to let the user explore the possible execution pathes. It is coupled with some GUI so the user can see the available LogicalSteps and select the one he wants to run.

2.4.3 Solver decider

In some case, the solver may have its own internal default strategy, this decider delegates the decision to the solver. For example, Timesquare default decider will select the LogicalStep that maximixe the number of MSE Occurence so the engine will execute as much DSE as possible.

2.5 MoCC/DSE compiler

This component is in charge of producing a TimeModel from the user model that conforms to the xDSML. Ie. a TimeModel that enforce the time constraints defined in the DSE and the MoCC.

2.6 Solver

The Solver is responsible for computing the possible LogicalSteps at execution step. It starts from a TimeModel associated to the user model.

2.7 Composition Operator applicator

This component doesn't exist yet in the GEMOC Studio. Its purpose is to allow the combined use of several xDSMLs and their associated MSE Engines.

3. MSE Engine details

The Execution Platform has been built using a component architecture that allows to easily switch implementation for a given component and customize the internal behavior of the engine.

In order to be taken into account by the Execution Engine, all the parts provided by the GEMOC process implement interfaces. We focus on the MSE Engine because it plays a central role in the platform and the other components collaborates with it.

3.1 Service view

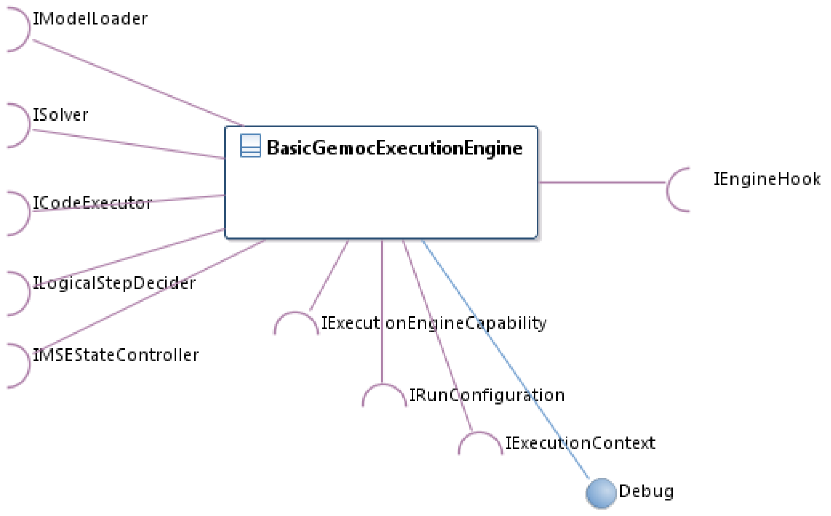


Figure 3.1: Services of the MSE Engine

Figure 3.1 shows which services are concretely handled by the MSE Engine. This engine is materialized by the class `BasicGemocExecutionEngine` in the sources.

It is possible to group the services depending on the intended user:

- Services for the Language Engineer. Such as : IModelLoader, ISolver, ICodeExecutor. They are built in the Language Workbench when defining the Domain Model, the Concrete Syntax, the DSA, the MoCC specific to the xDSML.
- Generic services, such as ILogicalStepDecider and IMSEStateController. They are provided in the Language.
- Configuration services, such as IExecutionEngineCapability, IRunconfiguration and IExecutioncontext. They are used to modularize the engine supported feature and to configure it.
- Services for the System Engineer. Such as IEngineHook. The are built in the Modeling workbench.

3.1.1 Mandatory services

This section presents the services that are required by the MSE Engine.

3.1.1.1 IModelLoader

This service allows to load the user model in a suitable way for both model execution and model animation. We need this service since the animation implementation may introduce some additional loading constraints. For example, this is currently the case for Sirius based Animator.

3.1.1.2 ISolver

This service provides a way to use different solvers. It is responsible for computing the possible LogicalSteps at a given point of time.

3.1.1.3 ICodeExecutor

The component that implements the ICodeExecutor is the entity responsible for executing the Domain-Specific Actions referenced by the occurrences of MSEs. These actions can act on the execution data. This CodeExecutor also has the role of giving an access to the DSA operations to the Engine.

Dependency inversion is especially important here in order to be able to run the Language Engineer's code from the engine.

3.1.1.4 ILogicalStepDecider

This services offers to decide which LogicalStep must be executed when at a given execution step, the solver proposes more than one possible LogicalStep for execution.

3.1.1.5 IMSEStateController

This services is used to help control the solver by forcing or not some MSE. This is typically used to provide a custom feedback from DSAs.

3.1.1.6 IRunConfiguration

This service gives some configuration information to the engine about how it should run.

3.1.1.7 IExecutionContext

This service is actually a provided service which provides some informaiton about the current internal state of the engine execution.

3.1.2 Optionnal services

These services are optional. They are mostly used to control or observe the engine internal behavior.

3.1.2.1 IExecutionEngineCapability

This API is used to enable or disable internal features of the MSE Engine. It mainly offers introspection methods so the engine may be more efficient when disabling some time consuming feature. This is typically used to enable or disable Trace generation.

3.1.3 Debug

This feature is currently not implemented as a service but should work as such. This Debug feature allows to notify a model that a given EObject is being accessed by a DSA. The current implementation uses Sirius service mecanism for this notification.

3.1.3.1 IEngineHook

When a component is provided with this API, the MSE Engine will call methods corresponding to its lifecycle. This is typically used to provide lightweight GUI defined directly in the Modeling Workbench because the user may want a dedicated GUI for his model. (Ie. a GUI more specialized than the animator provided by the xDSML)

3.1.4 Dependency inversion

One of the important role of the service architecture, is to offer dependency inversion so the MSE Engine remains generic. For example, the MSE Engine component cannot be dependent on a given component implementing the DSA for a language.

3.2 Execution flow

For now, we will consider that a typical usage scenario for the Execution Engine is to run a few steps of execution. We assume that the Execution Engine has somehow been connected and displays its execution's information to some user interface(s). This configuration step is describe later in Chapter 4.

In Figure 3.2, we show a typical scenario.

3.2.1 Initialization

The initialization of the engine consists in:

1. Generating the Time Model and FeedBack Model. This is realized by the MoCC/DSE compiler.
2. Initializing the Solver with the Time Model generated earlier.
3. Initializing the components that will provide the required services as seen in Section 3.1. Then connect them to the MSE Engine. The model will be loaded during this phase in a way that enable both execution and animation.

3.2.2 One global step of execution

Roughly, one step of execution consists in the following:

1. Query the Solver in order to get a list of possible LogicalSteps for this step.
2. Decide which logical step must run according to the decision policy that is currently configured.
3. Process the optional result of the previous MSA(s) into the Solver.
4. Execute the Model-Specific Action(s) associated to the Model-Specific Events found.

3.2.2.1 Query the Solver

Using the ISolver API, the Engine can request a scheduling trace from the Solver. This scheduling trace contains occurrences of MoccEvent instances, or ticks on the constrained clocks defined in the Solver.

3.2.2.2 Decide occurence

From this scheduling trace, the engine asks to a Decider which LogicalStep must be executed. Some version of the Decider may provide a GUI to ask the user or follow their own internal policy.

3.2.2.3 Process DSA feedback

If the previously run DSA executions have returned some values which have an impact on the future of the execution, then the engine may add some more constraints on the solver. It may force the presence or absence of some ticks for the next step.

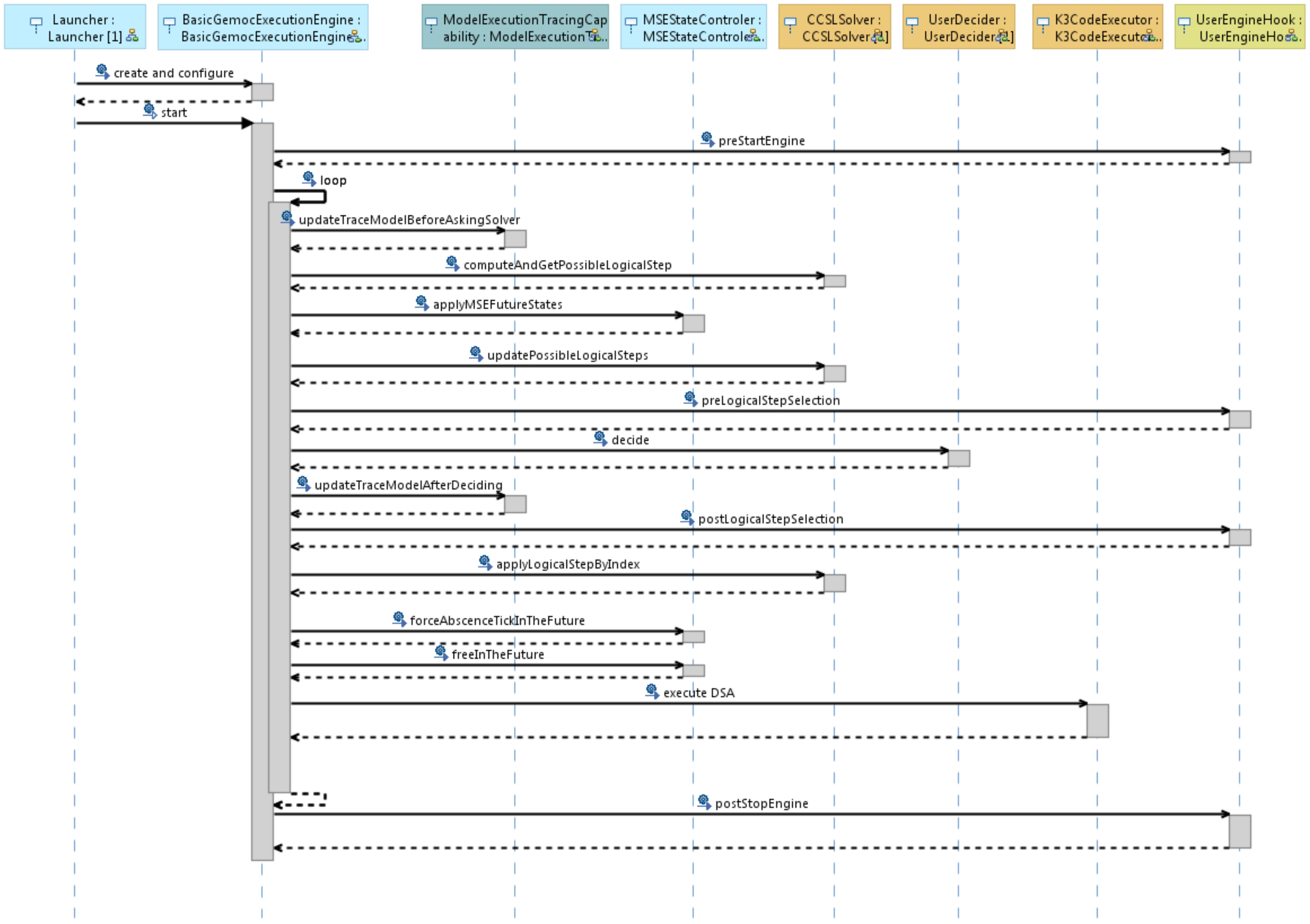

Launcher : Launcher [1]
BasicGemocExecutionEngine : BasicGemocExecutionEngine
ModelExecutionTracingCapability : ModelExecutionT...
MSEStateControler : MSEStateControle...
CCSLSolver : CCSLSolver[1]
UserDecider : UserDecider[1]
K3CodeExecutor : K3CodeExecuto...
UserEngineHook : UserEngineHo...
create and configure
start
preStartEngine
loop
updateTraceModelBeforeAskingSolver
computeAndGetPossibleLogicalStep
applyMSEFutureStates
updatePossibleLogicalSteps
preLogicalStepSelection
decide
updateTraceModelAfterDeciding
postLogicalStepSelection
applyLogicalStepByIndex
forceAbscenceTickInTheFuture
freeInTheFuture
execute DSA
postStopEngine

Figure 3.2: MSE Engine Sequence Diagram

3.2.2.4 Execute DSAs

These Model-Specific Events are handled by the ICodeExecutor which has the responsibility of executing the Model-Specific Actions contained by these MSEs.

3.2.3 Observe engine behavior and notification

During the execution, several elements can be observed. They are weaved in the execution flow.

Execution Data are serialized directly when they are modified by the DSA. The engine ensures that the DSA is executed in an EditingDomainTransaction so any connected Animator can receive the EMF notification for the change in the model.

If the engine is configured with the TracingCapability, it will save the trace model in each global step. Thanks to that dedicated views such as the Time Line View can present the history of the execution decisions.

In some situations, one may want to build a custom GUI for a dedicated model in a lightweight way. For such purpose, the engine offers a Hook service. If a hook has been configured, the engine will call the appropriate methods during its steps to notify the GUI.

4. Execution configuration

As the GEMOC Studio offers two workbenches, the Execution Engine can be configured from these workbenches. Obviously, the configurations possibilities won't be the same since they don't address the same user.

4.1 Configuring the engine from the Language Workbench

From this workbench, the Language Engineer will be able to define which component must be used for a given xDSML.

An xDSML must be packaged in an Eclipse bundle that offers the services required by the MSE Engine. When this bundle is deployed in a Modeling Workbench, the Execution Engine will be able to see it and use it with its associated components.

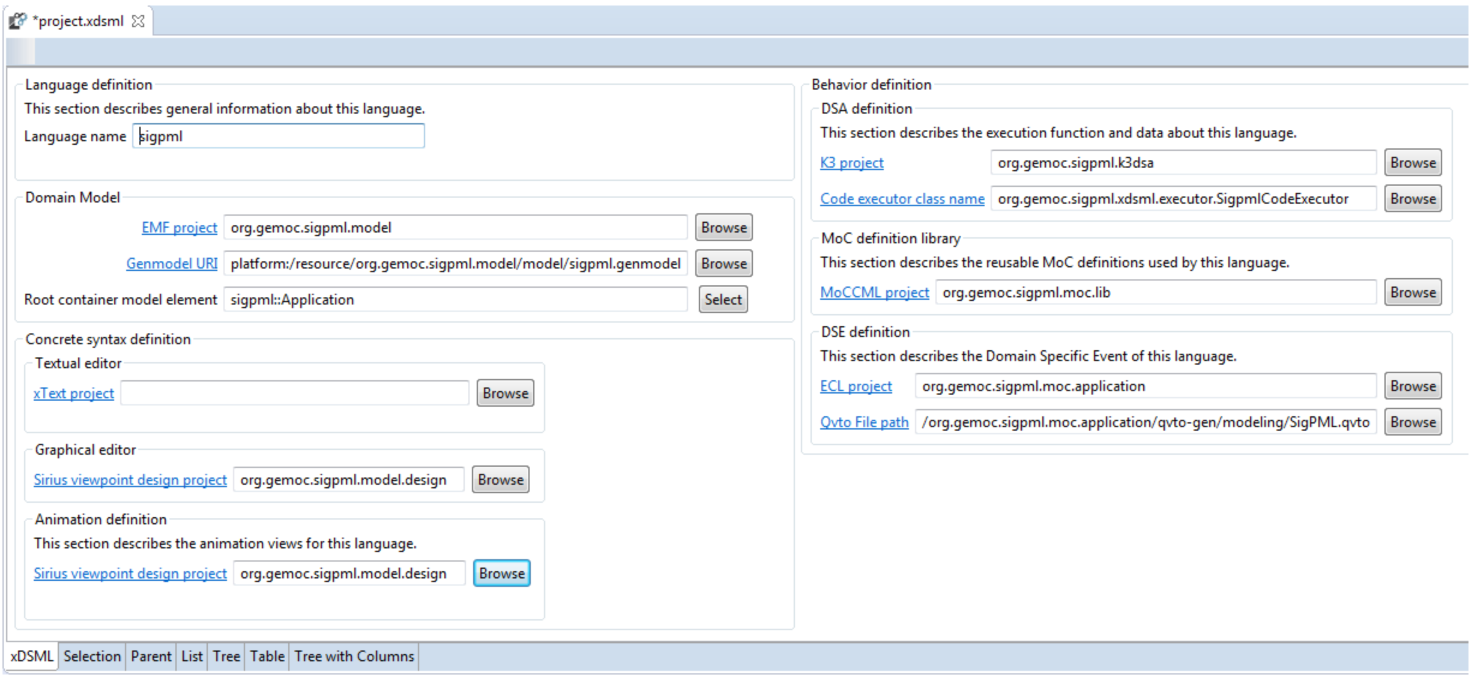
4.1.1 Configuring the engine with the basic xDSML editor

In the most commons scenarios, for a given xDSML, the language engineer will define a DSA, an Editor, an Animator, a DSE and a MoC. The Language workbench default process offers an editor to specify which project he want to use for each of these elements. A builder will take these information in order to automatically generate the appropriate service implementations. For example:

- a K3CodeExecutor is automatically configured to provide access to the DSA code;
- a CCSLSolver is generated from the MoC and the DSE definitions;
- a QVTo transformation is generated as MoCC/DSE compiler from the MoC and the DSE definitions;

The automatic generation assumes that the Language Engineer uses the proposed technologies.

Additionally, the builder will take care of some deployment point such as maintaining the project dependencies uptodate.



Language definition
This section describes general information about this language.
Language name: sigpml

Domain Model
EMF project: org.gemoc.sigpml.model
Genmodel URI: platform:/resource/org.gemoc.sigpml.model/model/sigpml.genmodel
Root container model element: sigpml::Application

Concrete syntax definition
Textual editor
xText project:
Graphical editor
Sirius viewpoint design project: org.gemoc.sigpml.model.design
Animation definition
This section describes the animation views for this language.
Sirius viewpoint design project: org.gemoc.sigpml.model.design

Behavior definition
DSA definition
This section describes the execution function and data about this language.
K3 project: org.gemoc.sigpml.k3dsa
Code executor class name: org.gemoc.sigpml.xdsml.executor.SigpmlCodeExecutor
MoC definition library
This section describes the reusable MoC definitions used by this language.
MoCCML project: org.gemoc.sigpml.moc.lib
DSE definition
This section describes the Domain Specific Event of this language.
ECL project: org.gemoc.sigpml.moc.application
Qvto File path: /org.gemoc.sigpml.moc.application/qvto-gen/modeling/SigPML.qvto

Figure 4.1: Configuring the Execution Engine via the xDSML editor

4.1.2 *Configuring the engine with the advanced xDSML editor*

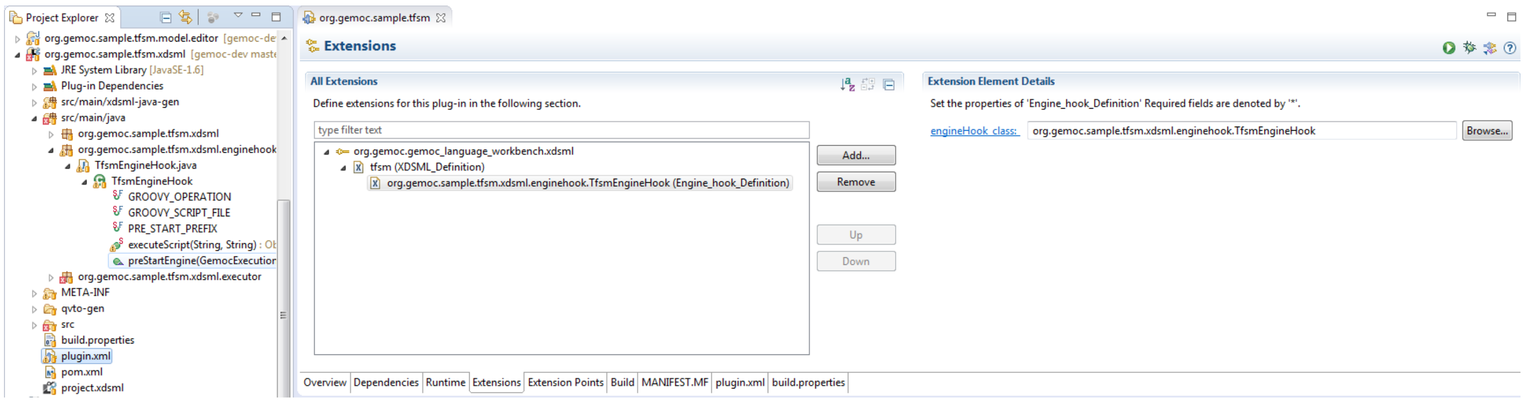
In more advanced scenarios, the Language Engineer will be able to replace the default technology by his own. In this case he will have to provide his own implementation of the services required by the MSE Engine. For example, this allows finely tune which Solver to use, or customize the access to DSA operations.

In Figure 4.1, the user has specified a custom CodeExecutor and a custom MoCC/DSE compiler (by changing the qvtopath). To use the default generated versions of these components, he would have simply leave these fields empty.

4.1.3 *Adding hooks to the engine*

The MSE Engine allows the Language Engineer to add some code that'll be called during the engine lifecycle. This is done by adding a class that implements the IHook API. See Figure 3.2 to know when the method of the class are called.

To tell the engine that a Hook class is available, the plugin.xml file must define an Engine_hook_Definition extension.



Define extensions for this plug-in in the following section.

- org.gemoc.gemoc_language_workbench.xdsml
  - tfsm (XDSML_Definition)
    - org.gemoc.sample.tfsm.xdsml.enginehook.TfsmEngineHook (Engine_hook_Definition)

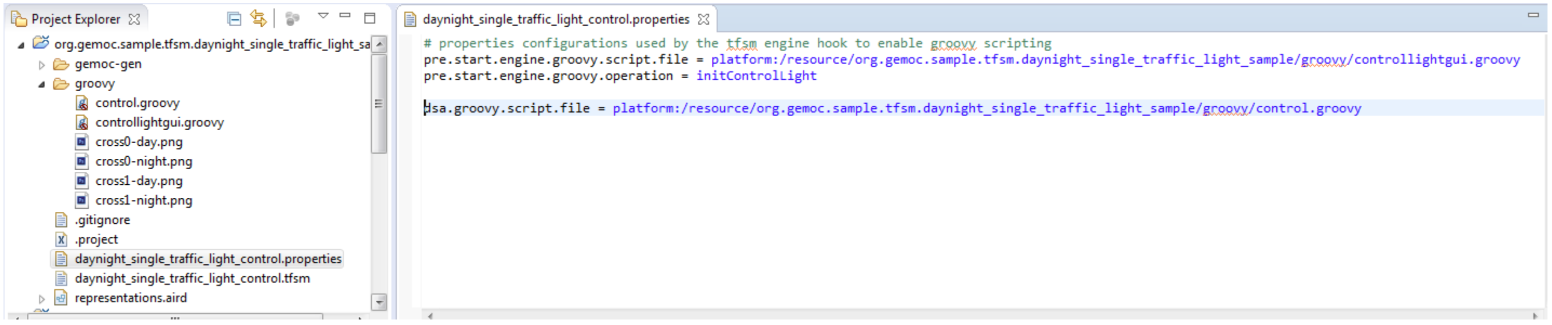
Set the properties of 'Engine_hook_Definition' Required fields are denoted by '*'.

engineHook_class: org.gemoc.sample.tfsm.xdsml.enginehook.TfsmEngineHook

Figure 4.2: Defining a hook class

In the TFSM example that is distributed in the Studio, the hook mechanism has been used to add a scripting support in the Modeling Workbench. This support allows to write some Groovy script code from the Modeling Workbench and run these scripts during the engine lifecycle. Figure 4.2 shows how this scripting support has been added to the engine for the TFSM language thanks to the Hook mechanism.

Figure 4.3 shows how this sample scripting support is used in the Modeling Workbench in order to open a GUI defined by the System Engineer for his model (in this case a Traffic light GUI).


Project Explorer
org.gemoc.sample.tfsm.daynight_single_traffic_light_sa
gemoc-gen
groovy
control.groovy
controllightgui.groovy
cross0-day.png
cross0-night.png
cross1-day.png
cross1-night.png
.gitignore
.project
daynight_single_traffic_light_control.properties
daynight_single_traffic_light_control.tfsm
representations.aird
daynight_single_traffic_light_control.properties

```
# properties configurations used by the tfsm engine hook to enable groovy scripting
pre.start.engine.groovy.script.file = platform:/resource/org.gemoc.sample.tfsm.daynight_single_traffic_light_sample/groovy/controllightgui.groovy
pre.start.engine.groovy.operation = initControlLight

dsa.groovy.script.file = platform:/resource/org.gemoc.sample.tfsm.daynight_single_traffic_light_sample/groovy/control.groovy
```

Figure 4.3: Example of scripting support implemented thaks to the Hook mechanism

4.2 Configuring the engine from the Modeling Workbench

From the Modeling Workbench, the System Engineer will be able to use the xDSML and associated configurations prepared by the Language Engineer.

In a Modeling Workbench the main way to configure the Execution Platform is via the Launch configuration.

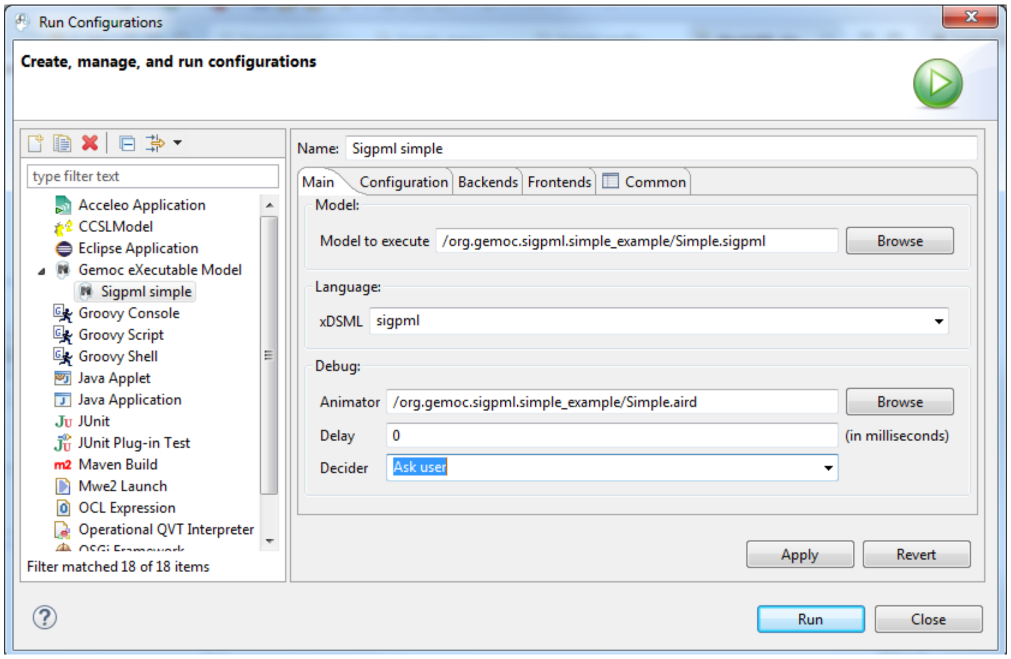
The Launch configuration mechanism is similar to the one used in Eclipse to run java programs for example, however the parameters are specific to GEMOC. Figure 4.4 shows the main tab of the GEMOC Launch configuration.

The first choice that has a significant impact on the engine is the selection of the xDSML language. This will tell the launcher to associate the engine with the components provided by the Language Engineer for this xDSML.

The second important choice is the selection of the Decider. This will configure the decision policy component that is in charge of selecting a LogicalStep when more than one LogicalStep is possible. Some Deciders such as *ask user* or *ask user (step by step)* may have a tight connection with the GUI.

Note: in the current version the selection of the Decider will be taken into account only when the launch is done in Debug mode. In Run mode, the decider is automatically a Solver Decider.

Running the Execution Engine in Debug mode rather than the normal Run mode also has some impact on the configuration. For example, the Animator will be enabled only in Debug mode.



Name: Sigpml simple

Model:
Model to execute /org.gemoc.sigpml.simple_example/Simple.sigpml

Language:
xDSML sigpml

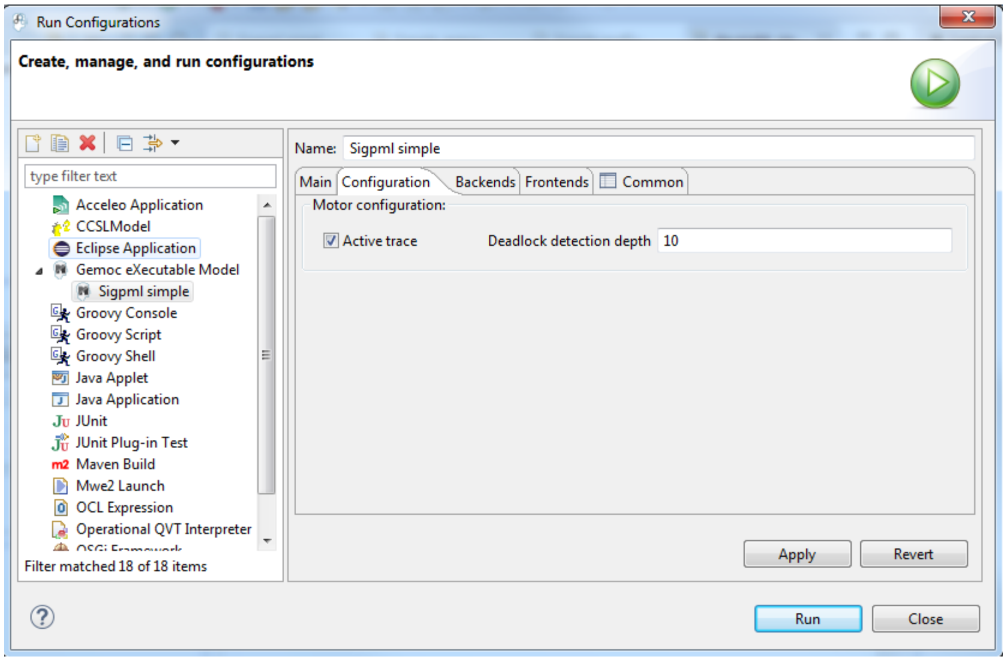
Debug:
Animator /org.gemoc.sigpml.simple_example/Simple.aird
Delay 0
Decider Ask user

Figure 4.4: Configuring execution engine via the launch configuration (main tab)

The second tab (see Figure 4.5) allows to enable or disable the Execution Engine tracing capability. If disabled, the TimeLine view will not be updated since it is based on these data.

The third tab (see Figure 4.6) allows to enable or disable some views dedicated for visualization. The Backends are displayers of runtime information made available by the engine. They are registered to the engine in order to listen for the information that the Execution Engine will record during the execution. Their task is mainly to represent the information sent by the Execution Engine. An Execution Engine can have as many Backends as needed, as well as no backend at all. Currently, only a Console view can be configured like this. In the future, some other visualization backends will be added.

The Frontend tab (see Figure 4.7) allows to automatically open the views that offers some interactions with the user. The Frontends are entities responsible for giving orders to the Execution Engine, typically through a User Interface (UI). They are connected to the Execution Engine directly or to a partial view of it Currently, it automatically opens the TimeLine view, the Engine Status view, the Logical Steps view and the Event Manager view.

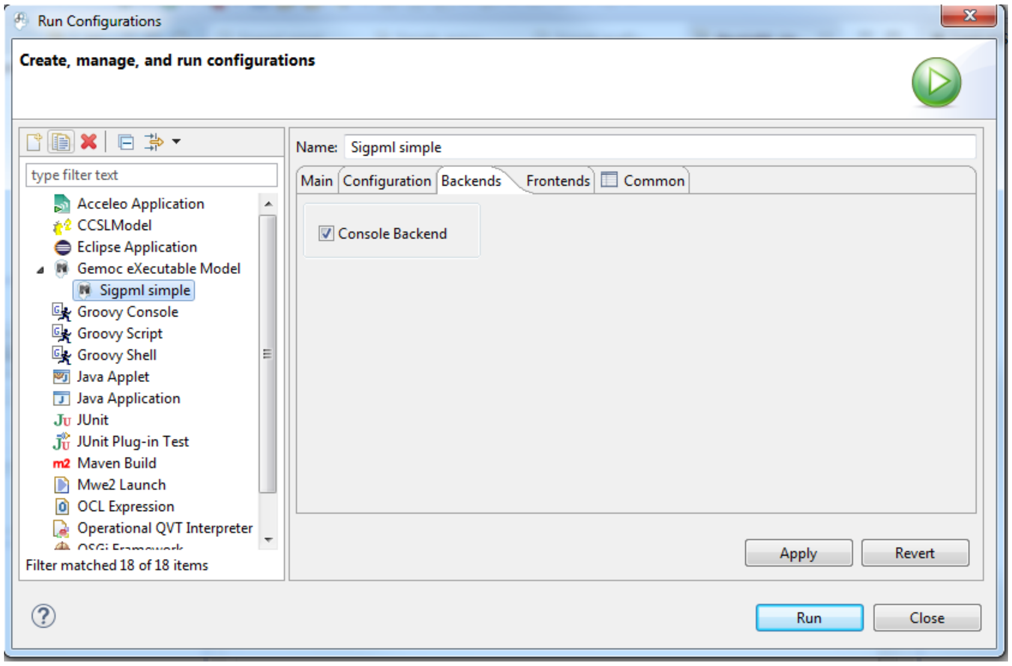


Name: Sigpml simple

Motor configuration:

☑ Active trace   Deadlock detection depth 10

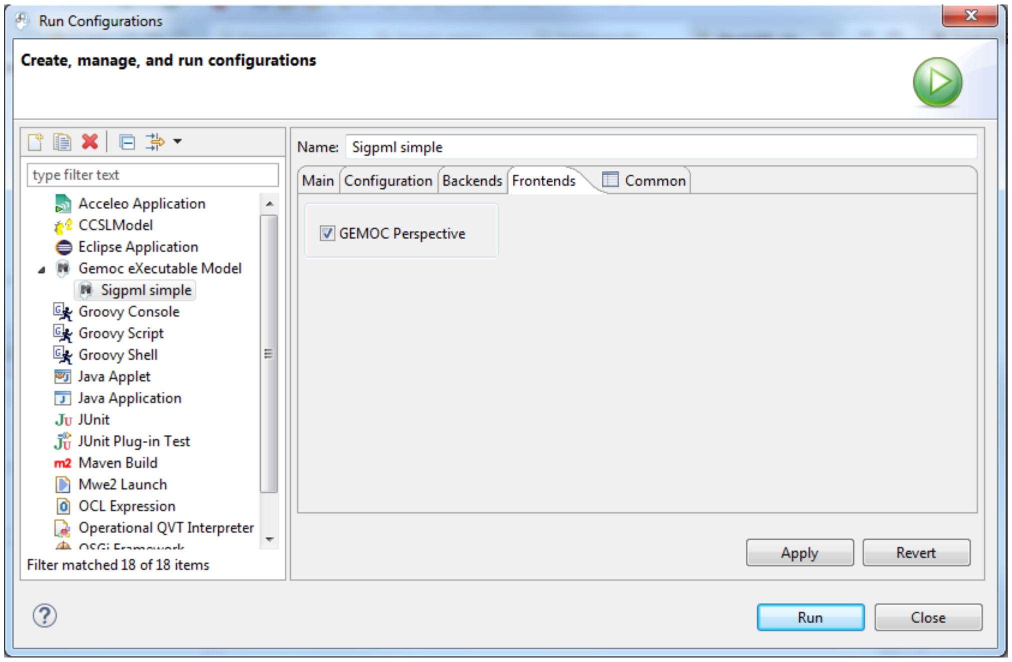
Figure 4.5: *Configuring execution engine via the launch configuration (configuraton tab)*



Name: Sigpml simple

☑ Console Backend

Figure 4.6: *Configuring execution engine via the launch configuration (backend tab)*



Name: Sigpml simple

☑ GEMOC Perspective

Figure 4.7: Configuring execution engine via the launch configuration (frontend tab)

5. Implementation and distribution

This chapter presents the current state of the engine and the implementations available. As shown by Figure 5.1, they are distributed directly within the GEMOC Studio.

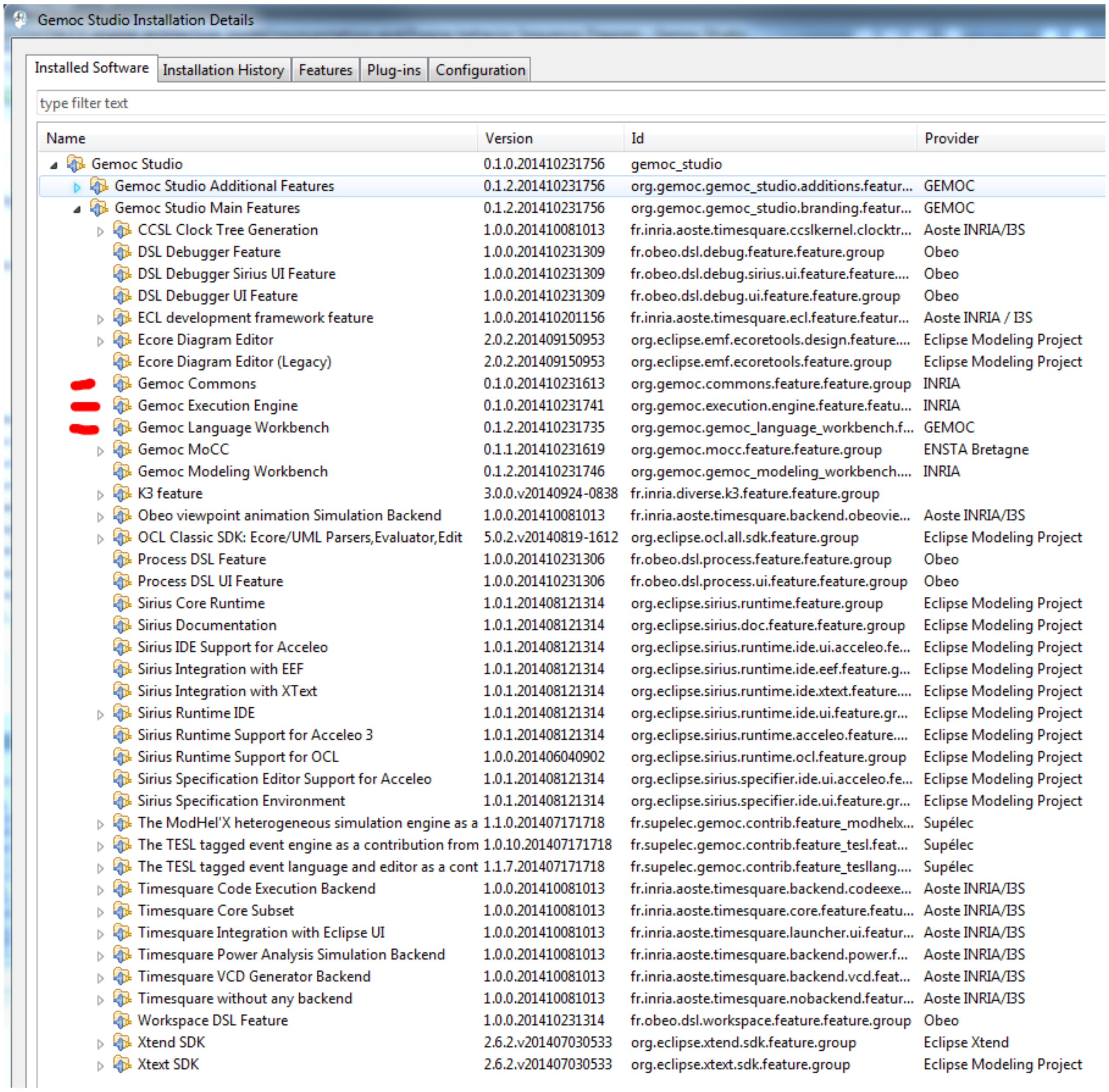
It is available for download from http://gemoc.org/studio/studio-download/

The features Gemoc commons, Gemoc Execution Engine and Gemoc Language Workbench contain all the plugins specific to the engine.

They also contains the service implementations for the Solver, the MoCC compiler, the DSE Compiler .

An User Documentation is available directly in the eclipse workbench. It describe the GUI of the various views provided in the Studio and includes the ones for the engine. (See Figure 5.2)

Another version of this doucmpentation is available online at the following address http://gemoc.github.io/gemoc-studio/. This online version correspond to the latest GEMOC Studio.

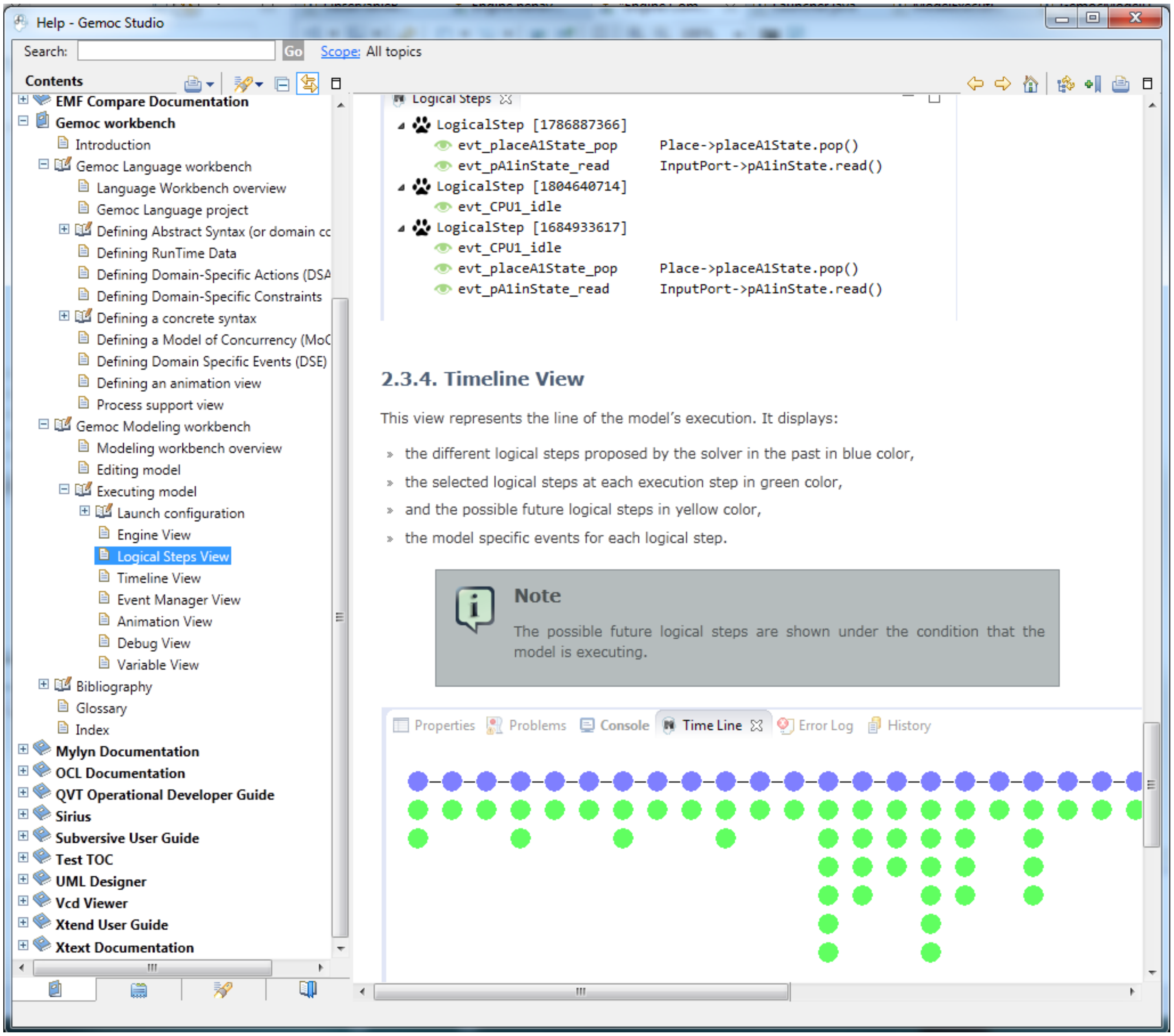

Gemoc Studio Installation Details

Installed Software | Installation History | Features | Plug-ins | Configuration

type filter text

Name	Version	Id	Provider
Gemoc Studio	0.1.0.201410231756	gemoc_studio	
Gemoc Studio Additional Features	0.1.2.201410231756	org.gemoc.gemoc_studio.additions.featur...	GEMOC
Gemoc Studio Main Features	0.1.2.201410231756	org.gemoc.gemoc_studio.branding.featur...	GEMOC
CCSL Clock Tree Generation	1.0.0.201410081013	fr.inria.aoste.timesquare.ccslkernel.clocktr...	Aoste INRIA/I3S
DSL Debugger Feature	1.0.0.201410231309	fr.obeo.dsl.debug.feature.feature.group	Obeo
DSL Debugger Sirius UI Feature	1.0.0.201410231309	fr.obeo.dsl.debug.sirius.ui.feature.feature....	Obeo
DSL Debugger UI Feature	1.0.0.201410231309	fr.obeo.dsl.debug.ui.feature.feature.group	Obeo
ECL development framework feature	1.0.0.201410201156	fr.inria.aoste.timesquare.ecl.feature.featur...	Aoste INRIA / I3S
Ecore Diagram Editor	2.0.2.201409150953	org.eclipse.emf.ecoretools.design.feature....	Eclipse Modeling Project
Ecore Diagram Editor (Legacy)	2.0.2.201409150953	org.eclipse.emf.ecoretools.feature.group	Eclipse Modeling Project
Gemoc Commons	0.1.0.201410231613	org.gemoc.commons.feature.feature.group	INRIA
Gemoc Execution Engine	0.1.0.201410231741	org.gemoc.execution.engine.feature.featu...	INRIA
Gemoc Language Workbench	0.1.2.201410231735	org.gemoc.gemoc_language_workbench.f...	GEMOC
Gemoc MoCC	0.1.1.201410231619	org.gemoc.mocc.feature.feature.group	ENSTA Bretagne
Gemoc Modeling Workbench	0.1.2.201410231746	org.gemoc.gemoc_modeling_workbench....	INRIA
K3 feature	3.0.0.v20140924-0838	fr.inria.diverse.k3.feature.feature.group	
Obeo viewpoint animation Simulation Backend	1.0.0.201410081013	fr.inria.aoste.timesquare.backend.obeovie...	Aoste INRIA/I3S
OCL Classic SDK: Ecore/UML Parsers,Evaluator,Edit	5.0.2.v20140819-1612	org.eclipse.ocl.all.sdk.feature.group	Eclipse Modeling Project
Process DSL Feature	1.0.0.201410231306	fr.obeo.dsl.process.feature.feature.group	Obeo
Process DSL UI Feature	1.0.0.201410231306	fr.obeo.dsl.process.ui.feature.feature.group	Obeo
Sirius Core Runtime	1.0.1.201408121314	org.eclipse.sirius.runtime.feature.group	Eclipse Modeling Project
Sirius Documentation	1.0.1.201408121314	org.eclipse.sirius.doc.feature.feature.group	Eclipse Modeling Project
Sirius IDE Support for Acceleo	1.0.1.201408121314	org.eclipse.sirius.runtime.ide.ui.acceleo.fe...	Eclipse Modeling Project
Sirius Integration with EEF	1.0.1.201408121314	org.eclipse.sirius.runtime.ide.eef.feature.g...	Eclipse Modeling Project
Sirius Integration with XText	1.0.1.201408121314	org.eclipse.sirius.runtime.ide.xtext.feature....	Eclipse Modeling Project
Sirius Runtime IDE	1.0.1.201408121314	org.eclipse.sirius.runtime.ide.ui.feature.gr...	Eclipse Modeling Project
Sirius Runtime Support for Acceleo 3	1.0.1.201408121314	org.eclipse.sirius.runtime.acceleo.feature....	Eclipse Modeling Project
Sirius Runtime Support for OCL	1.0.0.201406040902	org.eclipse.sirius.runtime.ocl.feature.group	Eclipse Modeling Project
Sirius Specification Editor Support for Acceleo	1.0.1.201408121314	org.eclipse.sirius.specifier.ide.ui.acceleo.fe...	Eclipse Modeling Project
Sirius Specification Environment	1.0.1.201408121314	org.eclipse.sirius.specifier.ide.ui.feature.gr...	Eclipse Modeling Project
The ModHel'X heterogeneous simulation engine as a	1.1.0.201407171718	fr.supelec.gemoc.contrib.feature_modhelx...	Supélec
The TESL tagged event engine as a contribution from	1.0.10.201407171718	fr.supelec.gemoc.contrib.feature_tesl.feat...	Supélec
The TESL tagged event language and editor as a cont	1.1.7.201407171718	fr.supelec.gemoc.contrib.feature_tesllang....	Supélec
Timesquare Code Execution Backend	1.0.0.201410081013	fr.inria.aoste.timesquare.backend.codeexe...	Aoste INRIA/I3S
Timesquare Core Subset	1.0.0.201410081013	fr.inria.aoste.timesquare.core.feature.featu...	Aoste INRIA/I3S
Timesquare Integration with Eclipse UI	1.0.0.201410081013	fr.inria.aoste.timesquare.launcher.ui.featur...	Aoste INRIA/I3S
Timesquare Power Analysis Simulation Backend	1.0.0.201410081013	fr.inria.aoste.timesquare.backend.power.f...	Aoste INRIA/I3S
Timesquare VCD Generator Backend	1.0.0.201410081013	fr.inria.aoste.timesquare.backend.vcd.feat...	Aoste INRIA/I3S
Timesquare without any backend	1.0.0.201410081013	fr.inria.aoste.timesquare.nobackend.featur...	Aoste INRIA/I3S
Workspace DSL Feature	1.0.0.201410231314	fr.obeo.dsl.workspace.feature.feature.group	Obeo
Xtend SDK	2.6.2.v201407030533	org.eclipse.xtend.sdk.feature.group	Eclipse Xtend
Xtext SDK	2.6.2.v201407030533	org.eclipse.xtext.sdk.feature.group	Eclipse Modeling Project

Figure 5.1: Distribution of the Execution Engine in the GEMOC Studio



EMF Compare Documentation
Gemoc workbench
Introduction
Gemoc Language workbench
Language Workbench overview
Gemoc Language project
Defining Abstract Syntax (or domain cc
Defining RunTime Data
Defining Domain-Specific Actions (DSA
Defining Domain-Specific Constraints
Defining a concrete syntax
Defining a Model of Concurrency (MoC
Defining Domain Specific Events (DSE)
Defining an animation view
Process support view
Gemoc Modeling workbench
Modeling workbench overview
Editing model
Executing model
Launch configuration
Engine View
Logical Steps View
Timeline View
Event Manager View
Animation View
Debug View
Variable View
Bibliography
Glossary
Index
Mylyn Documentation
OCL Documentation
QVT Operational Developer Guide
Sirius
Subversive User Guide
Test TOC
UML Designer
Vcd Viewer
Xtend User Guide
Xtext Documentation

```
Logical Steps
LogicalStep [1786887366]
    evt_placeA1State_pop      Place->placeA1State.pop()
    evt_pA1inState_read       InputPort->pA1inState.read()
LogicalStep [1804640714]
    evt_CPU1_idle
LogicalStep [1684933617]
    evt_CPU1_idle
    evt_placeA1State_pop      Place->placeA1State.pop()
    evt_pA1inState_read       InputPort->pA1inState.read()
```

2.3.4. Timeline View

This view represents the line of the model's execution. It displays:

- the different logical steps proposed by the solver in the past in blue color,
- the selected logical steps at each execution step in green color,
- and the possible future logical steps in yellow color,
- the model specific events for each logical step.

Note

The possible future logical steps are shown under the condition that the model is executing.

Figure 5.2: Engine user documentation in the GEMOC Studio eclipse help

6. Perspectives and issues

This chapter presents some elements that have been discussed and that may be of interest but that aren't available yet.

6.1 Replay Decider

The purpose of the Replay Decider is to replay a simulation by choosing the LogicalStep indicated in a trace model that was produced for example by a previous run.

Used for testing purpose, it should also raise an error when the LogicalStep which is indicated at a given point in the trace model is not among the LogicalStep allowed by the Model of Concurrency and Communication (the trace model is not compatible with the behavior of the model).

6.2 IModelLoder

In order to experiment how to load the model in a way that is suitable for animation, the current implementation is tightly coupled with Sirius animator. In future versions, it will use the IModelLoader API to release this coupling.

6.3 Debug

For the same reason as for the IModelLoader, in order to experiment how to add debug breakpoint and highlight the current element in the editor, the current implementation is tightly coupled with Sirius. In future versions, it will use a clearer API to release this coupling.

7. Conclusion

In this release, we have been able to build a generic Execution Engine which is able to execute models conforming to xDSMLs defined using the GEMOC softwares and methodologies.

Its flexible architecture is open and allow to run experiment with several different implementations of the different components.

The Execution Engine implementation and its architecture should be subject to some more changes before its next release as the composition of GEMOC xDSMLs will require the integration of new concepts and workflows.