Grant ANR-12-INSE-0011

ANR INS GEMOC

D4.2.1 - Generic Engine for Heterogeneous Models of Execution

Task 4.2.1

Version 0.2

DOCUMENT CONTROL

	–: 2013/11/28	A:	B:	C:	D:
Written by Signature	Florent Latombe				
Approved by Signature					

Revision index	Modifications
–	version 0.1 — Initial version
A	version 0.2 — Reworked most chapters
B	
C	
D	

Authors

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

Contents

1. Introduction

The GEMOC Execution Engine is a generic entity 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 Task 1.2.1 for detailed informations about the tools) and the corresponding methodology (see Task 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.

In this first version, we primarily focus on having a very generic workflow and architecture so as to be able to execute models conforming to any well-defined xDSML, in an implementation-technology-agnostic manner.

1.1 Perimeter

This document describes the v0 of the software deliverable D4.2.1 (*Generic Engine for Heterogeneous Models of Execution*). It presents the current state of the Execution Engine of the GEMOC Studio.

1.2 Definitions, Acronyms and Abbreviations

- **Model**: model which contributes to the convent of a View
- **Language Workbench**: a language workbench offers the facilities for designing and implementing modeling languages.
- **Language Designer**: a language designer is the user of the langage workbench.
- **Modeling Workbench**: a modeling workbench offers all the required facilities for editing and animating domain specific models according to a given modeling language.
- **Domain Engineer**: user of the modeling workbench.
- **GEMOC Studio**: Eclipse-based studio integrating both a language workbench and the corresponding modeling workbenches.
- **DSML**: Domain Specific Modeling Language.
- **xDSML**: Executable Domain Specific Modeling Language.
- **AS**: Abstract Syntax.
- **MOC**: Model of Computation.
- **RTD**: RunTime Data.
- **DSA**: Domain-Specific Action.
- **MSA**: Model-Specific Action, an instance of a Domain-Specific Action. While a DSA is specific to a language (to its Abstract Syntax), an MSA is specific to a model conforming to a language.
- **DSE**: Domain-Specific Event.
- **MSE**: Model-Specific Event, an instance of a Domain-Specific Event. While a DSE is specific to a language, an MSE is specific to a model conforming to a language.

- **GUI**: Graphical User Interface.
- **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.
- **API**: Application Programming Interface
- **CCSL**: Clock-Constraint Specification Language.
- **TESL**: Tagged Events Specification Language.
- **Static semantics:** Constraints on a model that cannot be expressed in the metamodel. For example, static semantics can expressed as OCL invariants.
- **Execution semantics:** Define when and how elements of a language will produce a model behavior.
- **Dynamic semantics:** see *Execution semantics*.
- **Behavioral semantics:** see *Execution semantics*.

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.

A well-defined 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 Computation (MoCs) (*cf.* WP2), and a FeedbackPolicy which specifies the influence of the return values from the Domain-Specific Actions on the future of the execution.

1.3.2 Implementation issues

However, all the different technologies which can be used to implement the above-mentioned language units were not made with the intent to interact with each other. Therefore, 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.

1.3.3 Available Implementations

During the early life of the Execution Engine, we are limited in the number of implementation technologies we have to be able to accommodate for. Therefore, we focus on the few pairs of technologies we know best to support first. However, this does not prevent us from designing the engine in the most generic manner so that when we want to add new implementation technologies (or later on, if users want to do that themselves), minimal changes are required.

Here are the technologies we plan to support during the development of the GEMOC Execution Engine:

1.3.3.1 MoC Implementations

- The GEMOC MoC Language as defined in WP2.
- The TESL formalism used by ModHel'X.

1.3.3.2 DSAs Implementations

Note: for now we consider the technologies used for the Domain-Specific Actions and their RunTime Data to be the same, but so far nothing tells us that this should be mandatory.

- Kermeta 2, provided by INRIA
- Kermeta 3, provided by INRIA
- Java with the EMF API

1.3.3.3 DSE Implementations

- At first we will only support ECL, provided by the Aoste Team from I3S.
- We plan to replace ECL with a more generic and tailored-to-our-needs language (cf D1.3.1)

1.3.4 Input

By itself, the engine is inert and needs to be driven by some form of input. Chapter 3 explains the mechanism of Control Panel that we intend to implement.

1.3.5 Output

The Execution Engine will provide an API so that external entities can exploit runtime information, like the scheduling trace or the execution trace, or the sequence of Domain-Specific Events that is being executed. Chapter 3 will also present the Backend mechanism that we intend to implement.

2. Architecture of the engine

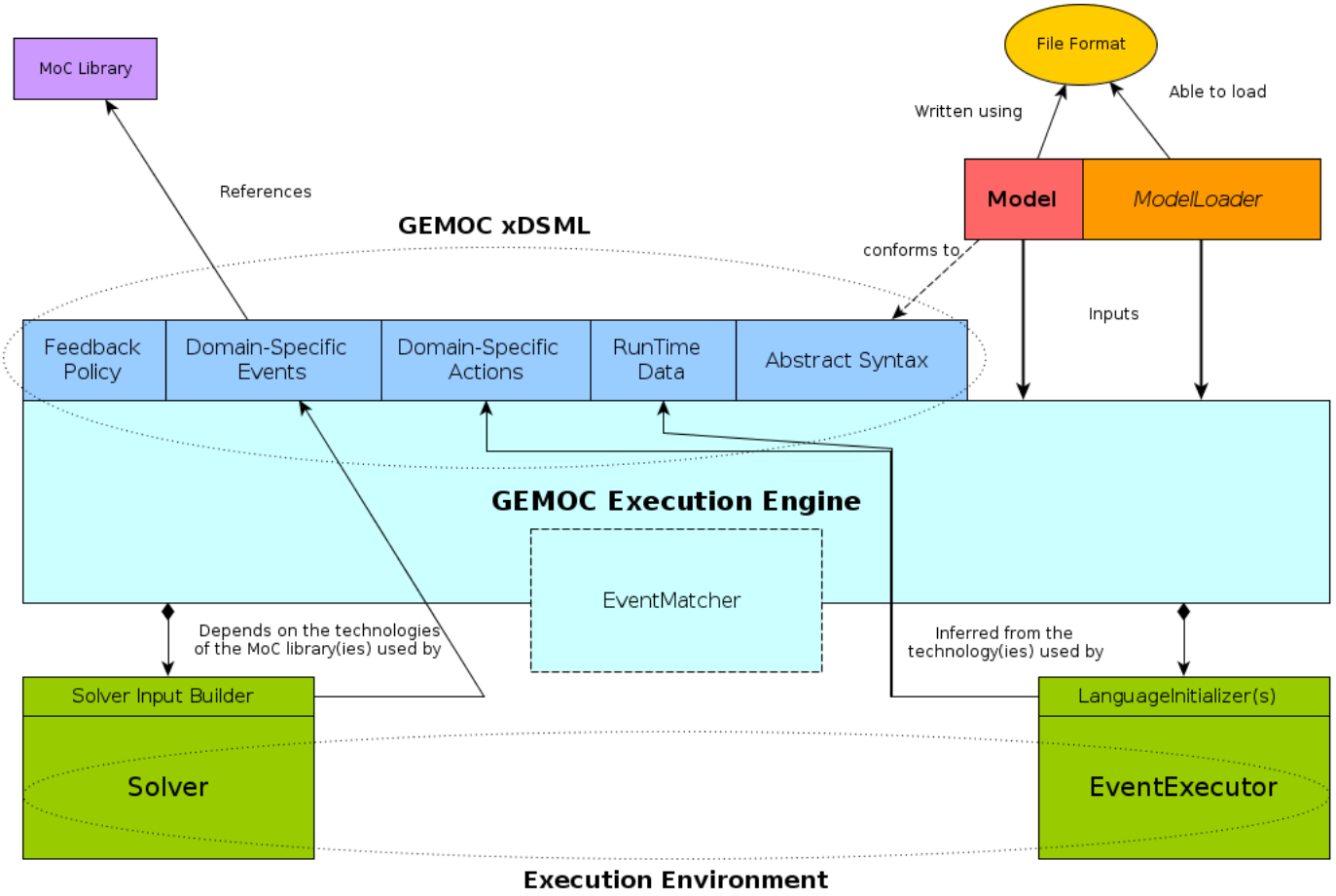

Figure 2.1: Architecture of the GEMOC Execution Engine

The architecture of the GEMOC Execution Engine is given informally on figure 2.1, and its corresponding UML Class Diagram is given on figure 2.2. The GEMOC Execution Engine requires the following components:

- An Execution Environment
- A well-defined GEMOC xDSML
- A model conforming to the above selected xDSML
- A ModelLoader utility that is able to load the above selected model

All these components are described in the sections below.

2.1 Execution Environment

An Execution Environment consists in a Solver and its associated Solver Input Builder and an EventExecutor which may provide some LanguageInitializer facilities. Execution Environments can be common to several xDSMLs as they are not

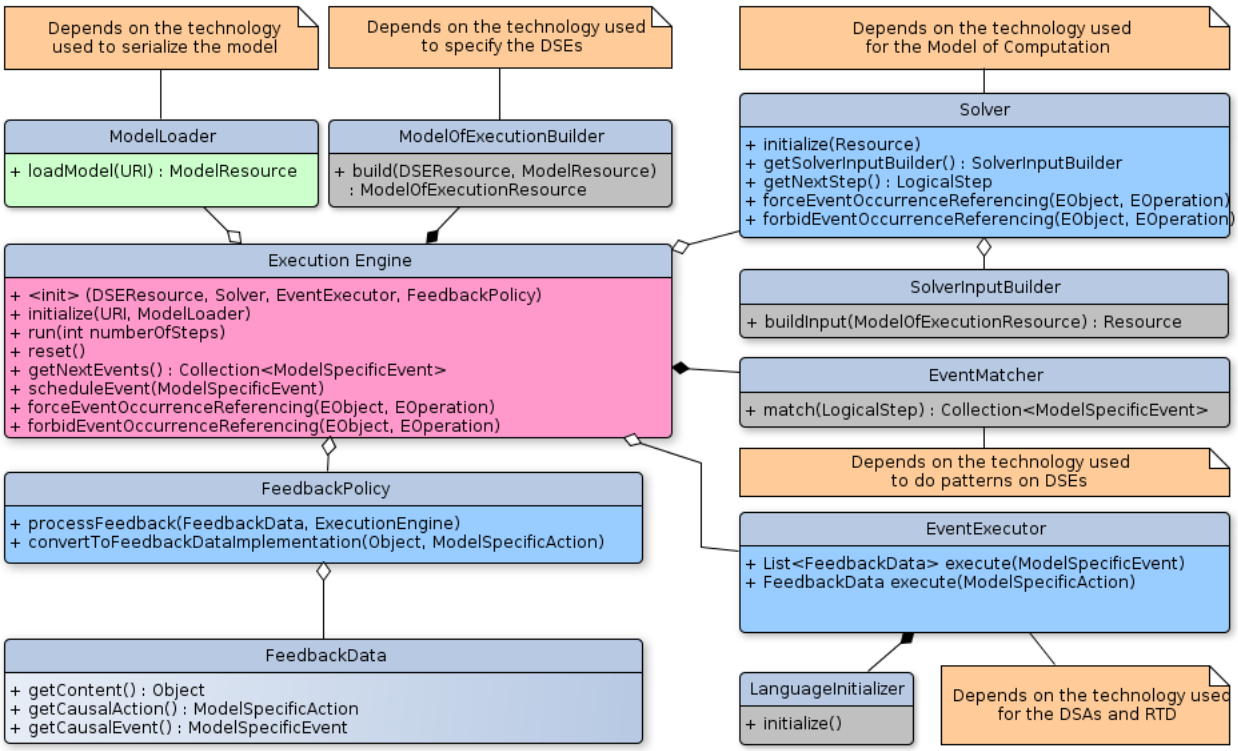

Figure 2.2: Class Diagram of the GEMOC Execution Engine

domain-specific ; they are instead MoC and DSA-technology specific since they depend on the languages used for the MoC Library(ies) and for the RunTime Data and Domain-Specific Actions. Therefore, it is possible for an entity outside of the Execution Engine to infer the Solver and EventExecutor to use depending on the nature of the implementation technology used for the RTD, DSAs and MoCs used by the xDSML. In particular, that is one of the roles of the tooling defined in Task 1.2.1.

2.1.1 Solver

The Solver takes for input a Solver Input file that is obtained thanks to its Solver Input Builder, the Domain-Specific Events of the language and the Model being executed. In CCSL, it is (roughly) a set of clocks and constraints between these clocks. The Solver is able to return through an API a scheduling trace conforming to the trace metamodel defined by I3S/Aoste Team in the RT-Simex project, upon request from the Solver.

2.1.1.1 Solver Input Builder

The Solver Input Builder is an entity attached to a Solver, responsible for transforming the Model of Execution into provide a Solver Input understandable by the Solver used.

2.1.2 EventExecutor

The EventExecutor is the entity responsible for executing the Domain-Specific Actions referenced by the occurrences of instances of Domain-Specific Events.

2.1.2.1 LanguageInitializer

The LanguageInitializer is an entity attached to an EventExecutor which is responsible for initializing whichever global variables the language used by the RTD/DSAs might require.

2.2 Generic Workflow

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 backend(s), but the details of this can be found in the next chapter (See 3).

2.2.1 Initialization

The initialization of the engine consists in:

1. Generating the Model of Execution and the Solver Input thanks to the Solver Input Builder, the Domain-Specific Events specification file and the model.
2. Initializing the Solver, using the Solver Input generated earlier.
3. Initializing the EventExecutor (including the LanguageInitializers if there are some).
4. Memorizing the Domain-Specific Events and Domain-Specific Actions (language-level data structure) and their instances to the model level (available in the Model of Execution) : Model-Specific Events and Model-Specific Actions.

2.2.2 One step of execution

One step of execution consists in the following:

1. Querying the Solver in order to get a Scheduling Trace for this step.
2. Matching the MoC-level events in order to find Model-Specific Events happening for this step.
3. Executing the Model-Specific Action(s) associated to the Model-Specific Events found.
4. Feedbacking the optional result of the MSA(s) into the Solver.
5. Recording everything all the while so that the Backends can exploit what is happening in the execution for animation, debugging, representation, etc... purposes.

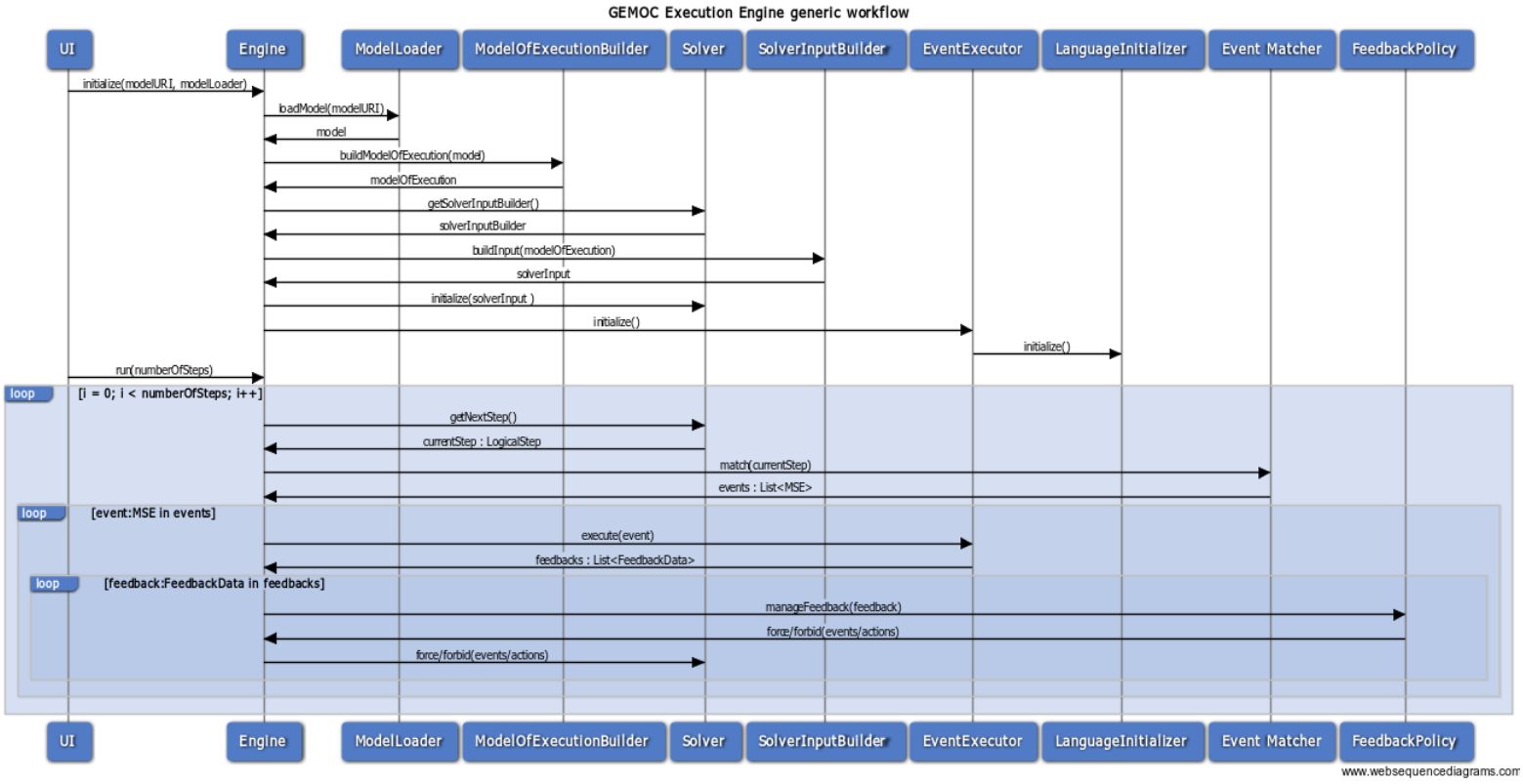

Figure 2.3: Sequence Diagram of initializing the GEMOC Execution Engine and running a few steps.

2.2.2.1 Querying

Using the Solver API, the Engine can request a scheduling trace from the Solver that conforms to the trace previously defined by Aoste/I3S in the RT-Simex project. This scheduling trace contains occurrences of MoC-level events, or ticks on the constrained clocks defined in the Solver Input.

2.2.2.2 Matching

From this scheduling trace, the engine uses its Event Matcher facility to execute some pattern matching on the trace (possibly with past scheduling traces too) in order to deduce which Model-Specific Events are happening at this step of execution. Corresponding Model-Specific Event instances are created.

2.2.2.3 Executing

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

2.2.2.4 Feedbacking

If the MSAs return some values which have an impact on the future of the execution, then the return value must be wrapped into an understandable data structure (FeedbackData implementation used by the FeedbackPolicy provided with the language) and interpreted by the FeedbackPolicy. This summons calls on the Solver API to add/remove constraints on the clocks managed by the Solver. *Note: The specific workflow of this step has not been frozen yet.*

2.2.2.5 Recording

All the while, everything happening must be recorded so as to be used by the Backends of the Execution Engine.

2.3 Execution within an Environment

In reality, we also want to be able to deal with reactive systems. The production of instances of Domain-Specific Events may come from another language or from the user through a UI. An example of sequence diagram is given in figure 2.4

For now, we have not precisely decided how we will handle such external events. Further versions of this document and its associated software should refine this issue.

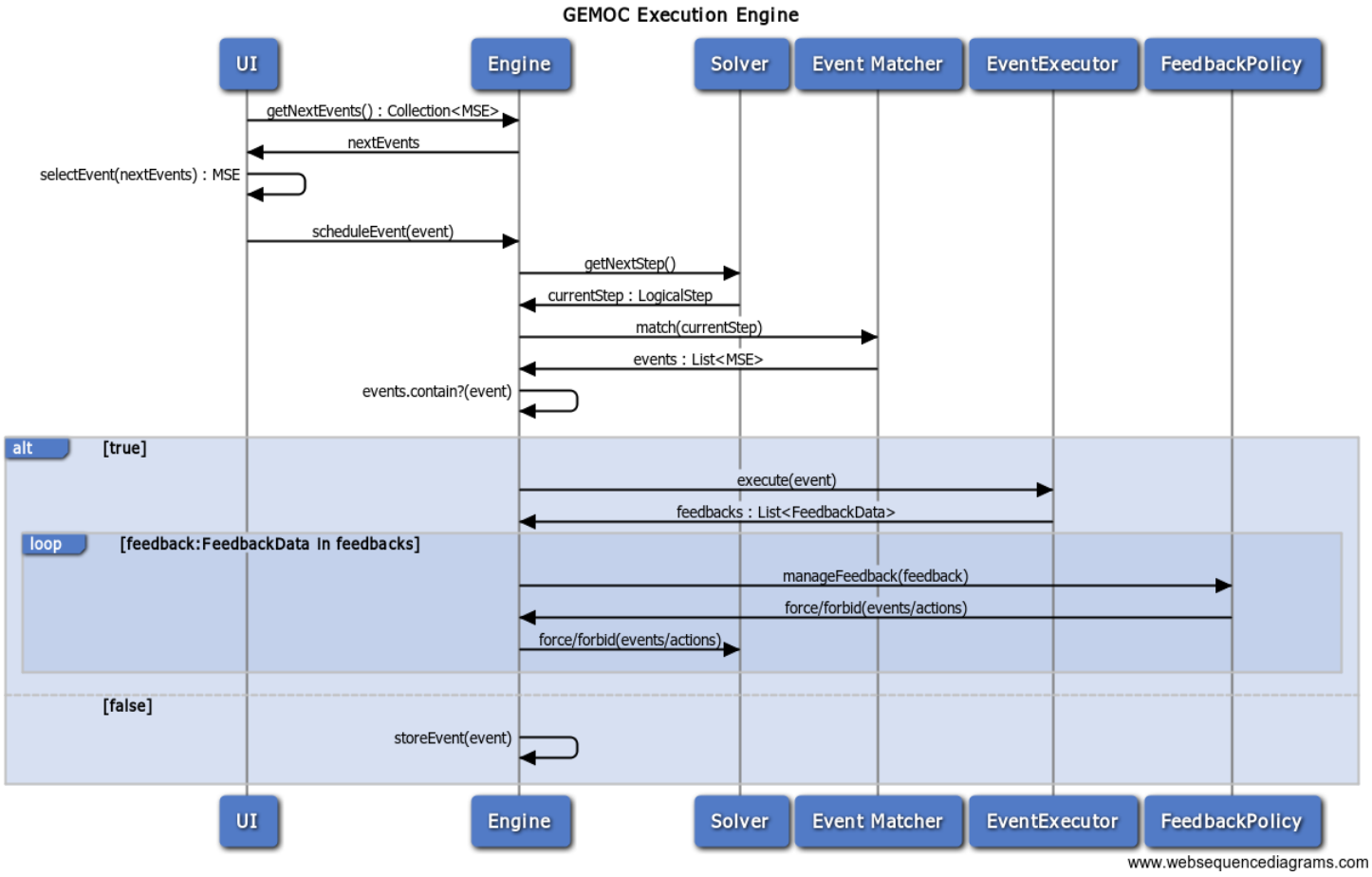

Figure 2.4: Sequence Diagram of the GEMOC ExecutionEngine

3. Connecting the engine

3.1 Engine Manager

The Engine Manager is an entity responsible for instantiating the engine with the components it is given through parameters and for connecting the engine to its Frontends and Backends. Its UML Class Diagram is given in figure 3.1. The API of the ExecutionEngine being potentially quite large, we have chosen to separate it into different interfaces which highlight the different kinds of Frontends we have identified.

3.2 Frontends

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 in order to guide the implementation of Frontends. An excerpt of the API of the Execution Engine is provided in figure 3.2

The API is not stable yet, and not complete, but we plan to provide at the least a mirror of the Eclipse Debug API for our Execution Engine.

We have identified two major types of Frontends, as described in the below subsections.

3.2.1 ControlPanel

A ControlPanel drives the engine by having access to the part of the API that makes the Engine execute a step forward, backward, pause, or play a certain number of steps. Thus, it is domain-agnostic as it literally has no idea about the domain it is controlling.

3.2.2 ScenarioBuilder

Either in batch or in interactive mode, the ScenarioBuilder can create and play a scenario. This means the ExecutionEngine must be able to control what input it is given by the scenario, as the MoC sets restriction on the scheduling of the Domain-Specific Events.

3.3 Backend

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.

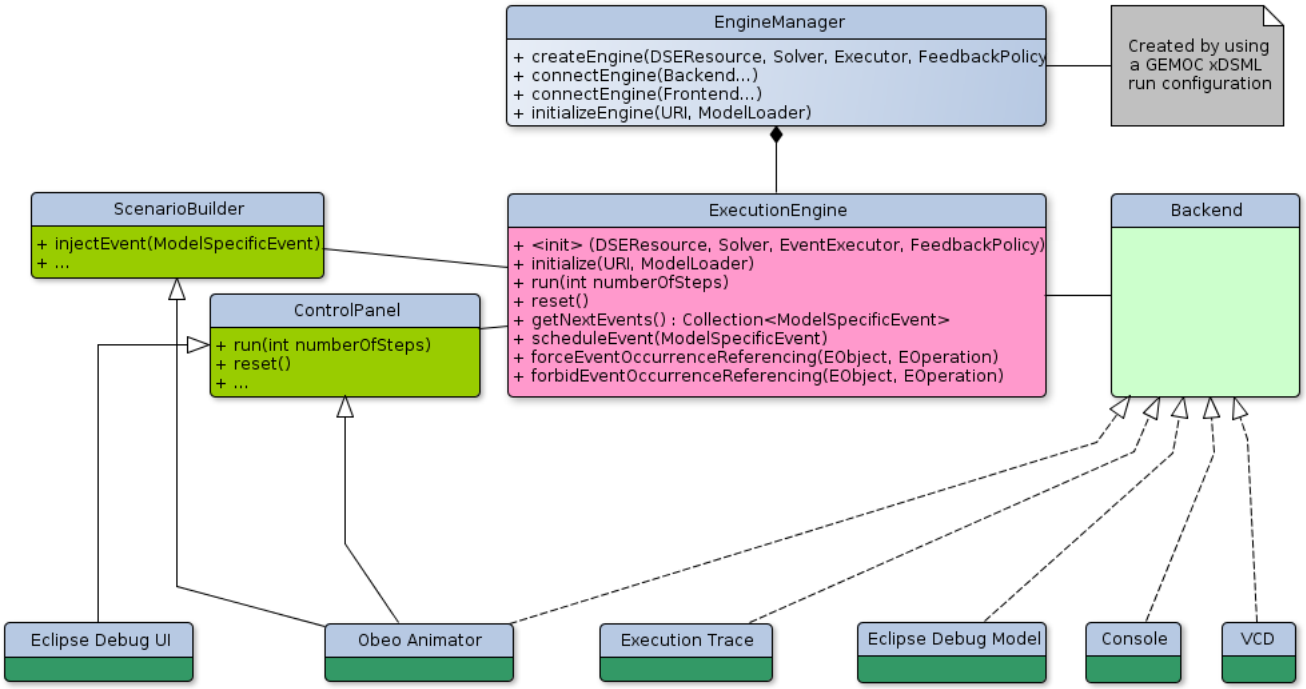

Figure 3.1: Class Diagram of the Engine Manager

```
/**
 * Runs the engine for a given number of steps.
 */
public void run(int numberOfSteps);

/**
 * Resets the engine and its components to its initial state.
 */
public void reset();

/**
 * Query to retrieve the MSEs which can be triggered by the environment, in no
 * particular order.
 */
public Collection<ModelSpecificEvent> getNextEvents();

/**
 * Tries to run one step of the execution corresponding to an occurrence of the
 * given mse, which should come from the getNextEvents() result.
 */
public void scheduleEvent(ModelSpecificEvent mse);
```

Figure 3.2: Excerpt from the API of the Execution Engine

4. Current Implementations

This chapter presents the current state of the engine and the implementations available.

4.1 Solver implementations

The Solver implementations must comply with the API used by the Execution Engine. They must also include a facility that is able to create, from the Model of Execution (instanciation of the Domain-Specific Events to a Model), the correct entry format for the solver (loosely named Solver Input). It must also be able to provide a scheduling trace conforming to the trace metamodel defined by I3S / Aoste team in the RT-Simex project.

4.1.1 Timesquare implementation

The Timesquare project has a solver entity which is already able to provide a scheduling trace conforming to the trace metamodel from the RT-Simex project. Its entry file format is a .extendedccsl file generated from the DSE specification written using ECL. A small wrapper has been written to accomodate the Solver API used by the GEMOC Execution Engine and the API exposed by the CCSL Solver.

4.1.2 ModHel'X implementation

We plan to be able to support the ModHel'X solver that uses TESL as MoC library language.

4.2 EventExecutor implementations

The EventExecutor implementations must comply with the API used by the Execution Engine. They must be able to locate objects and methods in the DSA bytecode and execute the methods.

4.2.1 Default EventExecutor implementation

The default EventExecutor provided is able to find and execute methods in Kermeta 3 and EMF Java bytecode.

4.3 Event Matcher of the Engine

The Event Matcher of the Engine is able to create the Model-Specific Events matched on the scheduling trace returned by the solver. They are matched according to the patterns used in the Domain-Specific Events definition. This entity is not fully clear for now due to a confusion in the implementation technologies used (ECL and CCSL being very related). We plan to refine it in later versions.

4.4 Frontend implementations

We plan to provide various useful Frontend implementations, which can be used out of the box. Of course, by contributing to the sources of the GEMOC Studio, anyone will be able to implement new Frontends.

4.4.1 Control Panel implementation

We plan to provide an implementation of ControlPanel using the Eclipse Debug GUI. It will be possible to move forward or backward in the execution, etc... .

4.4.2 Scenario Builder implementation

The Scenario Builder will use the Execution Engine API in order to sequentially propose future possible events to the user. From this builder, a Scenario (basically a sequence of Model-Specific Events) will be saved and it will be possible to share it or play through it fully using the Scenario Player.

4.5 Backend implementations

We plan to provide several useful backend implementations.

4.5.1 OBEO Animator

An OBEO Animator will illustrate the changes on the model while an execution is happening.

4.5.2 Console Backend

A Console Backend will allow us to debug and reason more easily on the execution of models.

5. Conclusion

Based on the experience of the Timesquare engine, we have been able to architecture a generic Execution Engine which will be able to execute models conforming to xDSMLs defined using the GEMOC softwares and methodologies. We are confident that it is only a matter of time before we can experiment it with several different implementations of the different components. The Execution Engine implementation and its architecture should be subject to a lot of changes before its next release as the composition of GEMOC xDSMLs should require new concepts and workflows.

A. User Manual

A.1 Starting from scratch

A.1.1 Retrieving the sources and software

First, one should download the GEMOC Studio at the following URL: https://ci.inria.fr/gemoc/job/org.gemoc.gemoc_studio.root/ Then, one needs to get the sources used by cloning the gemoc-dev project (*confere* Inria Forge).

A.1.2 Setting up the GEMOC Studio

On the long term, the GEMOC Studio will be available with the Language Workbench and the Execution Engine already deployed inside so that this step is not required. However in order to be sure to have the latest Language Workbench and Execution Engine, importing the following projects from the git repository into the workspace is necessary:

- org.gemoc.gemoc_language_workbench.api
- org.gemoc.gemoc_language_workbench.conf.model
- org.gemoc.gemoc_language_workbench.conf.model.edit
- org.gemoc.gemoc_language_workbench.conf.model.editor
- org.gemoc.gemoc_language_workbench.documentation
- org.gemoc.gemoc_language_workbench.extensions.k3
- org.gemoc.gemoc_language_workbench.feature
- org.gemoc.gemoc_language_workbench.p2updatesite
- org.gemoc.gemoc_language_workbench.root
- org.gemoc.gemoc_language_workbench.sample.deployer
- org.gemoc.gemoc_language_workbench.ui
- org.gemoc.gemoc_language_workbench.utils

and

- org.gemoc.execution.engine
- org.gemoc.execution.engine.commons
- org.gemoc.execution.engine.feature
- org.gemoc.execution.engine.io
- org.gemoc.execution.engine.p2updatesite
- org.gemoc.execution.engine.root

Launch a runtime Eclipse using these plugins. Let's name this configuration "GEMOC Language Workbench".

Note: You may need to launch it with increased memory (in the VM arguments of the configuration: "-XX:MaxPermSize=768M") in case you encounter some *PermGenSpace* errors.

A.2 In the Language Workbench

In the GEMOC Language Workbench, we can create a language and deploy it for the Modeling Workbench. For now, we will use the already-developed languages such as TFSM by importing into the workspace the sources of the example. We will also import the sources of the modeling workbench in order to be sure we have the latest version. Import the following projects into your workspace:

- org.gemoc.gemoc_modeling_workbench.feature
- org.gemoc.gemoc_modeling_workbench.p2updatesite

- org.gemoc.gemoc_modeling_workbench.root
- org.gemoc.gemoc_modeling_workbench.sample.deployer
- org.gemoc.gemoc_modeling_workbench.ui

and

- org.gemoc.sample.tfsm
- org.gemoc.sample.tfsm.ccslmoc
- org.gemoc.sample.tfsm.design
- org.gemoc.sample.tfsm.ecldse
- org.gemoc.sample.tfsm.k3dsa
- org.gemoc.sample.tfsm.model
- org.gemoc.sample.tfsm.model.edit
- org.gemoc.sample.tfsm.model.editor

Check that in the project org.gemoc.sample.tfsm:

- In src/main/xdsml-java-gen, package *org.gemoc.sample.tfsm.xdsml.api.impl*, delete the files TfsmDSAExecutor.java and TfsmInitializer.java if they get generated
- Right click on plugin.xml → replace with → HEAD revision → OK

Launch a new Eclipse runtime with these plugins. Let's name this configuration "GEMOC Modeling Workbench".

A.3 In the Modeling Workbench

In the Modeling Workbench, we are able to create models conformed to our deployed TFSM language. As we already have some instances of TFSM and there are still some troubles with generating the Model of Execution and the Solver Input, only a few limited examples can be used.

For now, import the *org.gemoc.sample.tfsm*, *org.gemoc.sample.tfsm.dse*, *org.gemoc.sample.tfsm.instances* projects into your workspace.

In *org.gemoc.sample.tfsm*, right click on "project.xdsml". Select Run As → Run Configuration. Create a new Gemoc Executable Model launch configuration. You can now select a model, like TrafficControl.tfsm and a language, tfsm.

By hitting run, an EngineManager is created. This EngineManager is hardcoded to create one ControlPanel of type basic GUI (described later) and one backend of type console. Later on, the run configuration should provide the means for the user to select which ControlPanel and Backends to use. This EngineManager also creates an Execution Engine with the given language and chosen model, and connects it to the ControlPanel and to the Backend(s).

There are a few interesting displays to consider. They are visible on figure A.1 and explained below:

1. GUI embryo: This basic GUI implements the ControlPanel interface. It is meant only to expose the API of the Execution Engine that any GUI will be able to use. In particular, future work should include connecting the API to the Eclipse Debugging API and UI. We can either ask the engine to do a full step of execution, or we can choose which Model Specific Event to execute (name usage has yet to be implemented due to difficulties with ECL). By default, a full step of execution will execute all the Model Specific Events available (subject to change). We can also reset the engine and the solver to zero.
2. GEMOC Execution Engine console: a console used for debugging the engine, in particular it shows what it receives from the solver, which events it matches, what is executing and what is sent as feedback to the solver.
3. GEMOC Execution Engine Feedback console: Where the FeedbackPolicy can print things. If the result of a DSA is a String then it is printed there. If it is something else then it is interpreted by the Policy and we have some written trace in order to be able to debug the execution by knowing what each DSA returned.
4. GEMOC Execution Engine Input/Output console: This console is a backend for the engine, implementing the Backend interface. Backends are observers on the Engine, and this one is simply printing any string or list of string received. For now we only record which actions are sent from the ControlPanel and which Model Specific Events are executed, but it is possible to record anything happening in the engine during execution and display it through any backend.

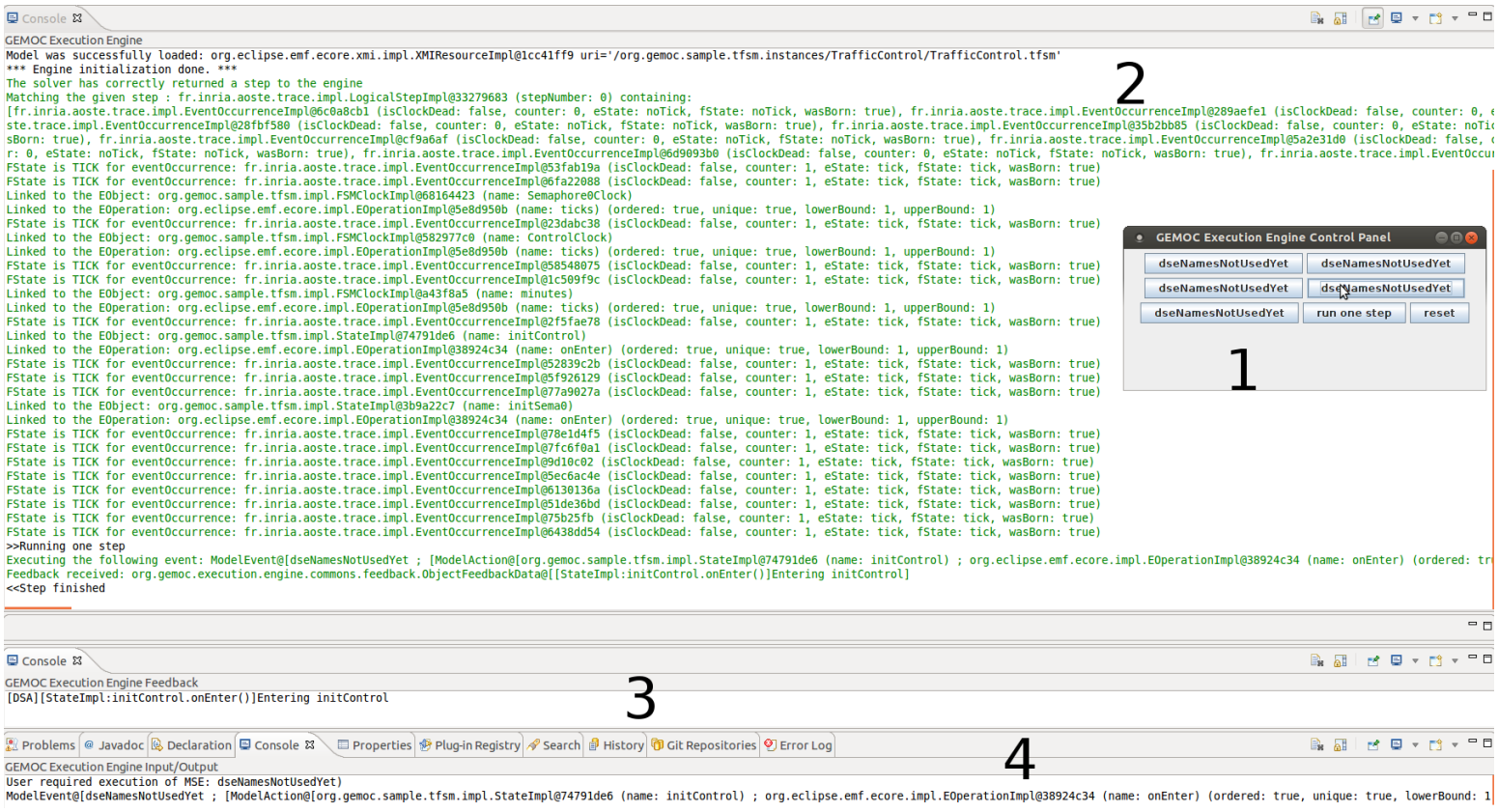

Figure A.1: Using the engine

