GEMOC Studio Guide

Francois Tanguy <francois.tanguy@inria.fr>,
Didier Vojtisek <didier.vojtisek@inria.fr>

GEMOC Studio Guide

Francois Tanguy <francois.tanguy@inria.fr>, Didier Vojtisek <didier.vojtisek@inria.fr>

Table of Contents

List of Figures

Introduction

The GEMOC Studio offers 2 main usages:

- Building and composing new executable DSML. This mode is intended to be used by language designers (aka domain experts).

- Creating and executing models conformant to executable DSMLs. This mode is intended to be used by domain designers.

Each of these usage has it own set of tools that are referenced as **Gemoc Language workbench** for the tools for *language designers* and **Gemoc Modeling Workbench** for the tools for *domain designers*.

Chapter 1. Gemoc Language workbench

1.1. Language Workbench overview

The GEMOC Language Workbench, intended to be used by language designers: it allows building and composing new executable DSMLs.
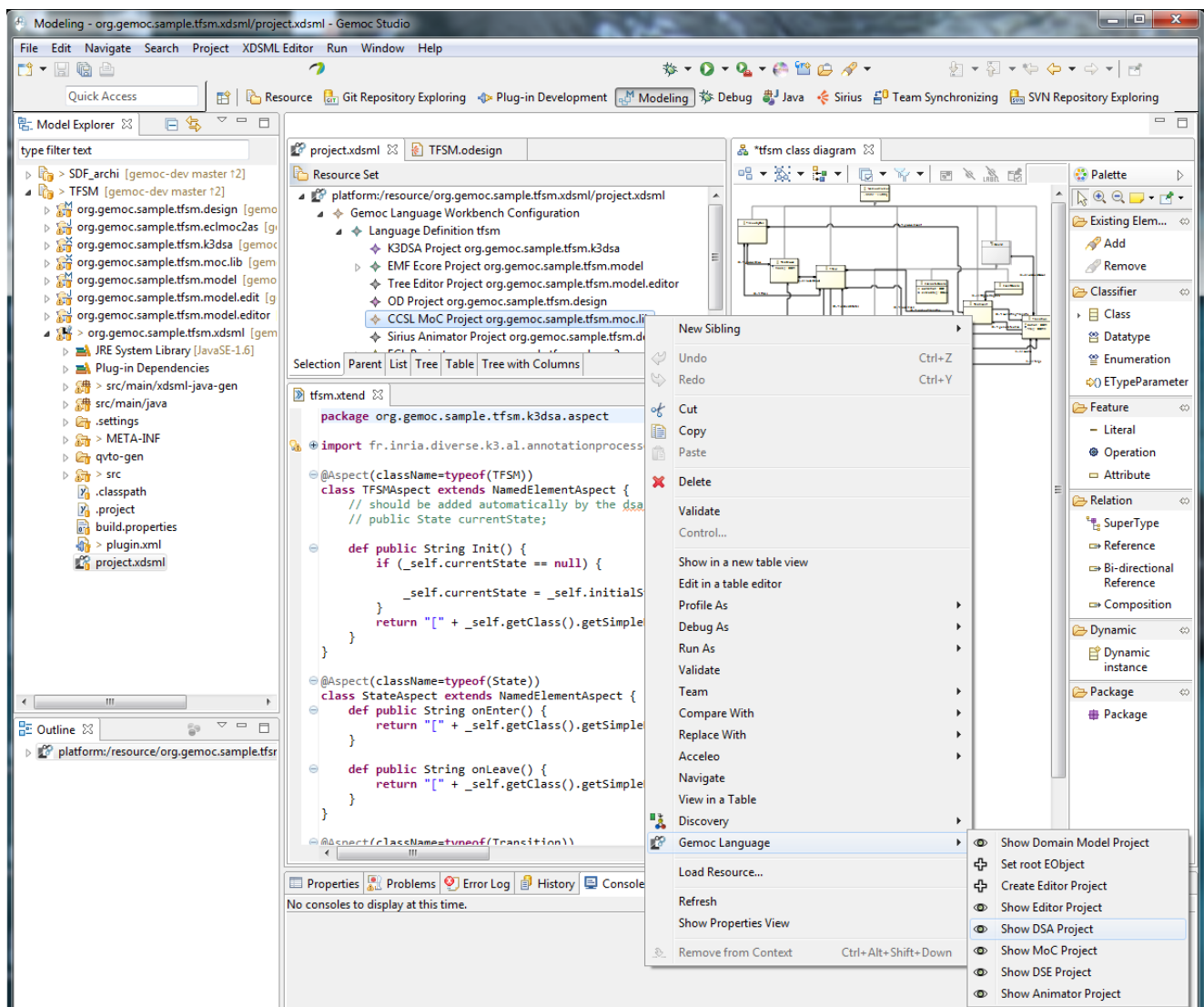
Figure 1.1. Screenshot of GEMOC Studio Language Workbench on the TFSM (Timed Finite State Machine) example.

1.2. Gemoc Language project

In the menu go to: File> New> Projects…> Gemoc Language project

This project will be the aggregator of the other Eclipse projects that will constitute the components of a Language Unit of the xDSML.

It mainly defines a project.xdsml file that references these other components.

The xDSML file isn't supposed to be edited manually. The pop up menu on the project (available on right click) or the Section 1.11, "Process support view" provides a set of wizard to fill this file.

From these information, the project will generate additional code that will be used by the execution engine when the language will be deployed and used by the Chapter 2, *Gemoc Modeling workbench*.

1.3. Defining Abstract Syntax (or domain concepts)

Defining the abstract syntax (AS) in the Language Workbench is done thanks to an EMF Ecore project that will be associated to the xDSML definition.

All ecore editors can be used to edit the concepts in the ecore file.

The EMF project can be created manually and then associated to the xDSML or directly created for the specific use of a given xDSML.

1.3.1. Method 1 : Creation via xDSML popup

To create the EMF project:

- right click on the xDSML project

- navigate in the *Gemoc Language* entry.

- click on *create Domain Model project*

This will open a wizard asking either to create a new EMF project or select an existing one (usefull for method 2). Creating a new EMF project will ask for some

> **Note**
>
> In future version of the studio, this method may disapear and be replaced by method 3.

1.3.2. Method 2 : Manual creation and association to the xDSML

From eclipse menu:

- *File > New > Projects… > Ecore Modeling Project*

- in the Ecore diagram, define the concepts for your language as a class diagram.

> **Tip**
>
> There are several other ways to create valid EMF projects. Amung them we can consider: *File > New > Projects… > EMF Project*, *File > New > Projects… > Empty EMF Project*, *File > New > Projects… > xText Project* … However, this isn't in the scope of this document to detail them or explain how to correctly configure them for being used by Gemoc.

Then associate the EMF project to the xDSML. The association is done either with right click on the xDSML project, *Gemoc Language > create Domain Model project > Select existing EMF project*; or via the Gemo Guideline view.

1.3.3. Method 3 : Creation or selection via Gemoc Guideline view

The Gemoc Guideline view offers an integrated vision of the most important steps of the xDSML creation process. When selecting the *Define Domain Model* task, a button allows to launch the wizards of method 1.

TODO new project wizard (via popup, via process) TODO select existing project TODO Ecore editor

1.4. Defining RunTime Data

1.5. Defining Domain-Specific Actions (DSAs)

In Gemoc, the DSAs are developed using Kermeta 3.

> **Note**
>
> There may exist other ways to create DSAs, but these methods aren't in the scope of this document.

- right click on the xDSML project

- navigate in the *Gemoc Language* entry.

- click on *create DSA project*

In the wizard you can select an existing templates to help create the aspects classes that will extend the classes of the Domain model.

You can manually create a Kermeta 3 project in the main eclipse menu, and then associate it to an xDSML project.

1.6. Defining Domain-Specific Constraints

1.7. Defining a concrete syntax

The xDSML can support different concrete syntaxes. Most EMF based editors should work however Gemoc provides additional support for some editors. Editors explictly supported are: EMF tree editor, xText editor, Sirius editor.

1.7.1. Defining a concrete syntax with xText

1.7.2. Defining a concrete syntax with Sirius

1.8. Defining a Model of Concurrency (MoC)

1.9. Defining Domain Specific Events (DSE)

1.10. Defining an animation view

The animation layer is an extension on top of a graphical editor defined with Sirius.

TODO Debug layer, Animation layer

1.11. Process support view

TODO present process view

Chapter 2. Gemoc Modeling workbench

2.1. Modeling workbench overview

The GEMOC Modeling Workbench, intended to be used by domain designers: it allows creating and executing models conformant to executable DSMLs.
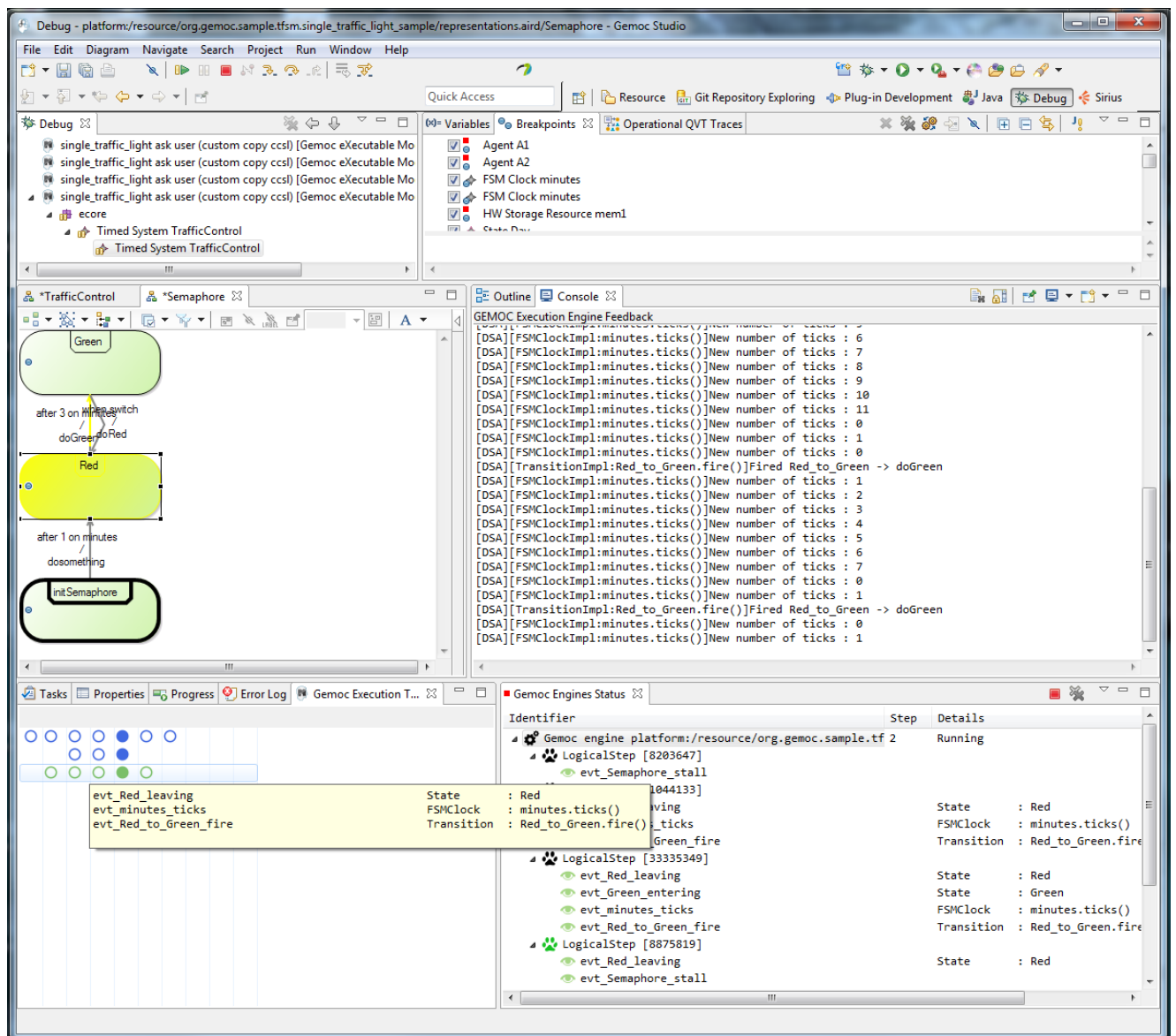
Figure 2.1. Screenshot of GEMOC Studio Modeling Workbench on the TFSM example (execution and animation).

2.2. Editing model

2.3. Executing model

2.3.1. Launch configuration

The Gemoc launch configuration offers both a Run and a Debug mode.

General options

- Model to execute : this is the model that will be run

- xDSML : this field allows to select among the valid variants of the executable language that are available for the model (Ie. the combinaison of DSA, DSE and MoCC that can be used on the given domain model)

- Decider : this field allows to select the solver strategy used by the engine when several Logical Steps can be triggered. Possible choice are :

  - Solver proposition : the solver internal stategy will be used to selecton Logical Step

  - Random : will randomly select one of the available Logical Step (warning: execution cannot be reproduced when using this Decider)

  - Ask user : (available only in Debug mode), this option will use the Logical Step View or the Timeline View to present the available Logical Steps and pause if there are more than one Logical Step. The user will then need to click on one of the Logical Step to continue.

  - Ask user (Step by step) : (available only in Debug mode), this option is similar to the previous one. However, it will pause on every Logical Step, even if there is only one Logical Step that can be triggered. This is more or less equivalent as putting a breakpoint on every MSE of the language.

More Deciders will be developped (for example for playing predefined scenario).

Run mode

In run mode, it offers the faster way to run the model. It cannot be paused. However, you can stop it. It offers a limited set of views :

- the Engine View allows to stop a running model.

- the Timeline View is displayed at the end of the execution in order to control the resulting execution trace.

If more feed back are required, please use one of the front end or back end available for the xDSML.

Debug mode

In debug mode, the engine offers more control on the execution. It allows to pause, add break point, and run in a step by step mode.

It reuses the Eclipse Debug perspective and some of its views and add some Gemoc specific views.

- the Engine View allows to stop a running model.

- the Timeline View is displayed during all the simulation.

- the Event Manager View is displayed during all the simulation.

- the Event Manager View is displayed during all the simulation. It can display both an animation layer and a debug layer.

- the Debug View. This view presents an interface for Step by Step execution at the Logical Step level or even at the DSA level.

- the Variable View. This view presents the Runtime Data as a (EMF based) tree.

When running a simulation in Debug mode, it is configured to activate automatically the Debug layer and the Animation layer in the Animation view.

Backends and frontends

Back ends and front ends offer additionnal view that can respectivly display informations from the running model or provide event input to the running model.

These backends and front ends usually open dedicated views. These views are always opened in all modes (Run or Debug).

2.3.2. Engine View

The engine view displays a list of execution engine and their statuses:

- its number of execution steps,

- its current running status,

- and its logical steps deciding strategy.

2.3.3. Logical Steps View

The logical steps view displays the list of possible future execution. This list is provided by the solver. This view is organized around a tree. For each logical step, its underlying events can be seen and possibly for each event the associated operation is visible.

> **i** **Note**
>
> This view displays nothing when execution runs in "run mode", per say this view is only of use when running in "debug mode".

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.

> **i** **Note**
>
> The possible future logical steps are shown under the condition that the model is executing.

In addition to displaying information, it also provides interaction with the user. During execution, it is possible to come back into the past by double-clicking on any of the blue logical steps. It does two things:

1. it resets the solver's state to the selected execution step,

2. and it resets the model's state to the selected execution step.

2.3.5. Event Manager View

2.3.6. Animation View

- Debug Layer

- Animation Layer

2.3.7. Debug View

This view is part of the Debug perspective. It presents an interface for Step by Step execution at the Logical Step level or even at the DSA level. When an execution is paused, this view presents the current Logical Step.

When paused on a Logical Step, the Step over command allows to go to the next Logical Step. The Step Into command allows to run separatly each of the internal DSA calls associated to the Logical Step.

2.3.8. Variable View

This view is available on the Debug perspective. When an execution is paused, this view presents the current Runtime Data as an EMF based tree.

Chapter 3. GEMOC xDSML definition tutorial

> **Warning**
>
> This tutorial is a work in progress and is not yet finished. It still contains TODO and must be polished.

3.1. Introduction

The purpose of this tutorial is to explain on a simple example how to define an xDSML (eXecutable Domain Specific Modeling Language). It is structured as follows. First, we recall the GEMOC approach, including architecture of the GEMOC xDSML and the main characteristics of a GEMOC process. Then, we present the overall process that will be illustrated by this tutorial. The next sections illustrate the process on a concrete example, the definition of automata. We describe the application domain, and then we split the development in several increments to illustrate different aspects of the GEMOC studio.

1. First implementation of the deterministic Automata xDSML with a focus on the DSA.

2. Second implementation with a focus on the MoCC (to show that a balance must be found)

3. Graphical visualization.

4. Nondeterministic automata (evolution of the previous example)

5. Pushdown Automata (demonstrate the feedback mechanism)

6. Integration of user defined code (connect the automata to a specific graphical interface).

7. Other increments:

   • Composite DSE

   • Composition operators

   • Traces

Conventions used in this tutorial

Normal text is used to explain the process and give some rationals and the proposed solutions.

This kind of paragraph corresponds to manipulation to be performed of the GEMOC studio, either the Language Workbench or the Modeling Workbench.

3.2. The GEMOC Approach for defining eXecutable DSML (xDSML)

3.2.1. Architecture of a GEMOC xDSML

> **ℹ️ Note**
>
> We consider that the Abstract Syntax (AS) of the DSML is already defined and thus that its design is not part of the GEMOC xDSML process. Nevertheless, the process could easily be extended with a new step which consists in defining the AS and its Concrete Syntaxes (CS).

Once the AS is defined — and possibly the CS ---, the execution semantics of the DSML can be defined. It includes the definition of:

- the **DSA** (Domain Specific Actions) which includes EF (Execution Functions) and ED (Execution Data): it defines the runtime data and the actions to handle them,

- the **MOCC** (Model of Concurrency and Communication) which deals with concurrency aspects,

- the **DSE** which maps DSA and MOCC,

- the **visualization** which provides views to control and monitor the execution of models conforming to the xDSML.

There is no mandatory order in which these different parts have to be built. Thus, a first description of the process can focus on this characteristic (see overview process). The xDSML parts may be built in any order. This process stresses a first step which consists in eliciting the requirements on the expected execution semantics. This elicitation is not always easy to achieve for the system engineers, thus we believe that providing samples of models as well as some scenarios describing their execution is a good way for system engineers to describe they expectations.

Furthermore, it is recommended to built the xDSML using increments and iterations. Increments address requirements the ones after the others. Iterations allow to rework the different components until the requirements are fulfilled.
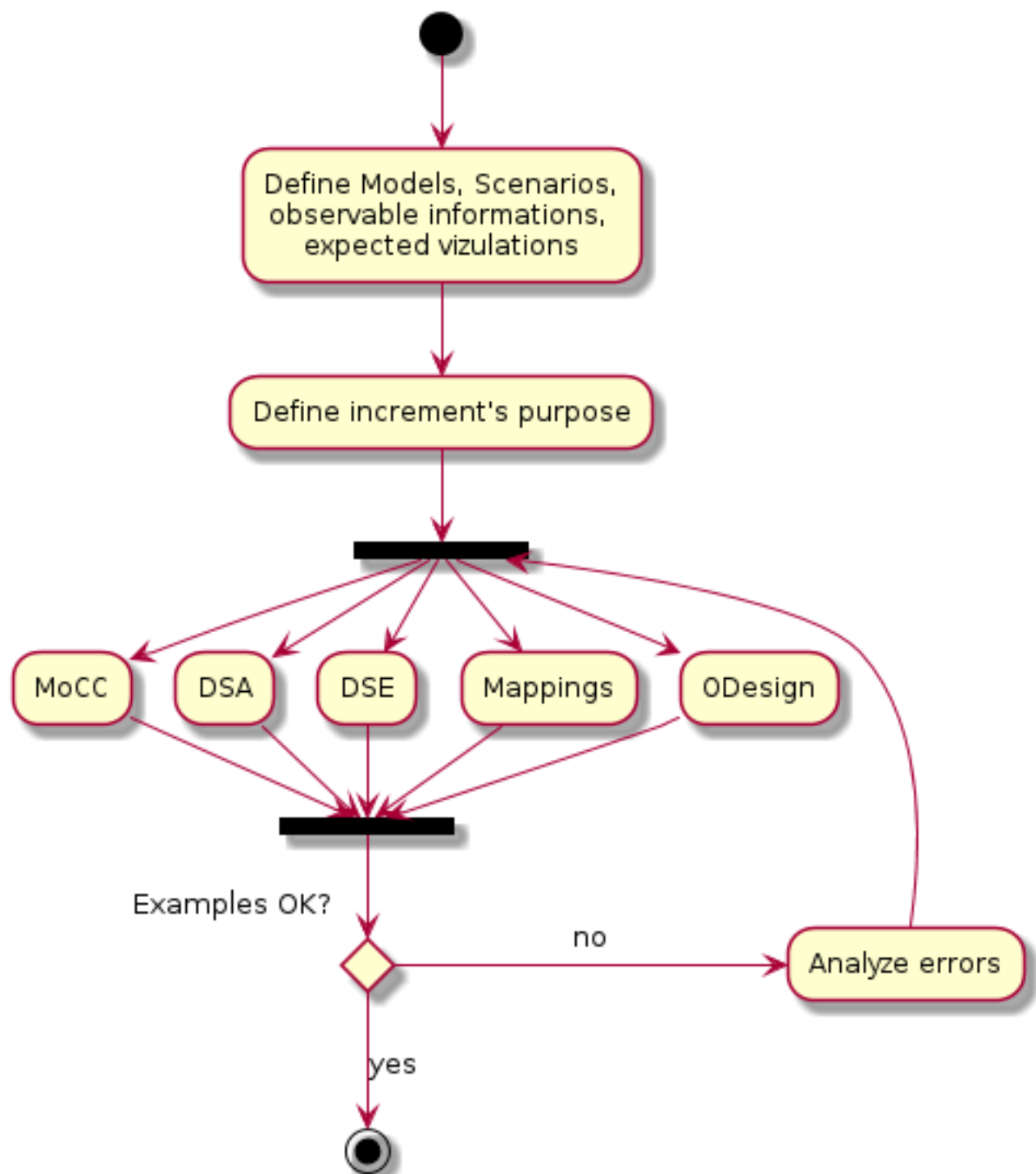
Figure 3.1. Overview of the GEMOC process

> **Note**
>
> For clarity, the iteration to choose the next increment is not represented of the diagram.

3.2.2. Main Characteristics of the GEMOC Process.

The main characteristics of the definition of an \xdsml according to \gemoc are the following:

- It is **user-oriented**. Indeed, we believe that to build the right xDSML, the end-users' expectations have to be handled in the first place. As GEMOC targets simulation of models, it is important to elicit the expectations of the end-users, i.e. the system engineers, in term of simulation of its models.

- It is **incremental**. All end-users expectations will not be handled all at the same time but we recommend to define several increments that will progressively includes these expectations seen as requirements. The purpose is to make the definition of the execution semantics simpler by gradually integrating the requirements, and thus the difficulties.

- It is **iterative**. For an increment, all the components will certainly not be done right and complete at the first time. Thus, iterations will be necessary to tweak the definitions of the xDSML. Examples provided by the end users will help in deciding whether an iteration is finished or not.

- It is **highly concurrent**. Indeed, the main steps can be run at the same time, possibly by different persons (even if it not always very easy using Eclipse). We can imagine that the DSE model and mappings can be first defined to specify the overall architecture of the xDSML semantics, including requirements on DSA as specification of the expected execution functions and requirements on the MoCC as expected MoCC events and relations on them. The MoCC libraries, the DSA implementation and the animator can then be developed at the same time, by different persons. Of course the DSE model can still be changed and the others updated.

3.2.3. Recommanded GEMOC Process

An xDSML is a language that targets a specific purpose. When defining such a language it is important to first identify that purpose so as to build the right language. Thus, we propose a methodology which focuses on the end-user expectations --- why models are animated? --- and then build the different parts of the xDSML to fulfill these expectations.

In the context of the GEMOC project, the main aim is to be able to animate and simulate heterogeneous models based on different xDSMLs. Thus the end-user expectations will encompass the different xDSML involved in the model to build. This global process which includes the dispatch of expectations on the different parts of a model will be addressed in a future revision of this document when all constituents will be further defined.

The main steps to achieve the definition of an xDSML according to the GEMOC approach are described hereafter. They are summarized as a UML activity diagram. Package notation is used to group activities by domain (and could be considered as a kind of compound activity).
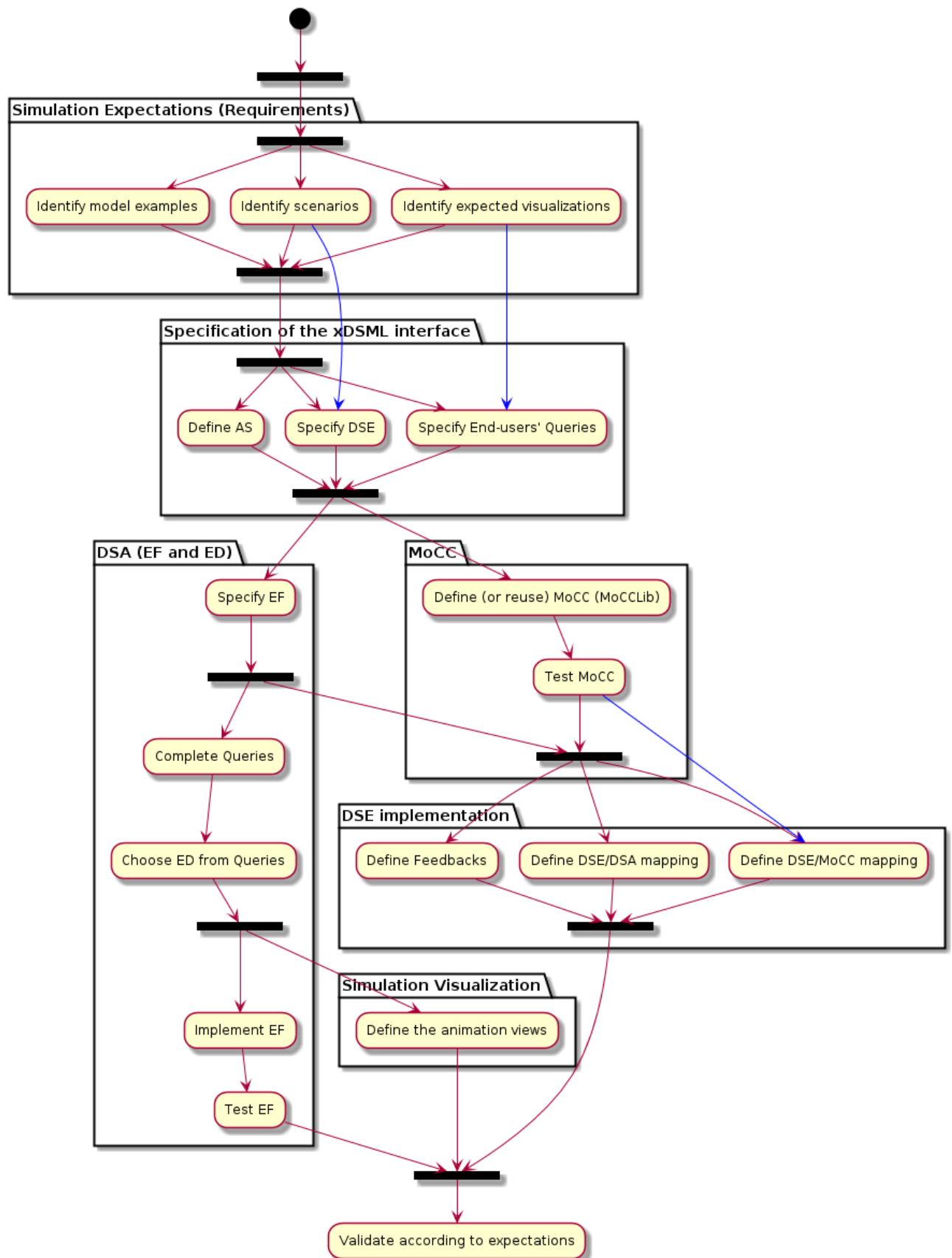
Figure 3.2. Recommended GEMOC Process

> ⚠️ **Warning**
>
> This process is the recommanded one, and the one used in this tutorial. Nevertheless, it is possible to define an xDSML with other processes depending on the background of the language designer, his knowledge of the GEMOC approach, the considered DSML, etc.

3.3. Definition of the requirements/expectations on the xDSML

The first step consists in describing the system engineers' expectation concerning the considered domain in terms of execution semantics and visualization at runtime. As it is often a complex task for system engineers to formally describe their expectations, we advocate to define them through examples to complement the informal descriptions.

An example is composed of:

1. A **model** which is conform to the DSML AS).

2. A **scenario** which describes a particular use of the model. A scenario is considered of events, that is stimuli that trigger evolution of the model.

3. **Expected results** while the scenario is played. Expected results include values of runtime data, possible next events, etc. They are way to describe the expected behavioral semantics.

Obviously, it is possible to share some elements between several examples. For example, the same model may be used by several examples.

This steps is important to understand the expectations of the system engineers. As providing a formal specification of his/her expectations is generally not easy, giving some examples including expected results on specific scenario is a good way to specify through examples.

Furthermore, these examples will be used to validate the implemented xDSML.

Finally they can be used to define the increments in the development of the xDSML by defining the set of models and scenarios each increment must handle.

3.3.1. Application Domain: Automata

We consider the domain of **automata**. An automaton may be used to specify a language defined on a set of symbols called alphabet. The following example shows an automaton which recognizes the language *a*ba**.

3.3.2. Description of automata

An automaton is composed of a finite set of states, transitions and symbols. An automaton must have exactly one initial state (and thus at least one state). One transition connects a source state to a target state and is labelled by a symbol. On the example, states are represented by circles, doubled-circled states are accepting states (or final states). A transition is depicted as an arrow from the input state to the output state. An arrow without input state points to the initial state.
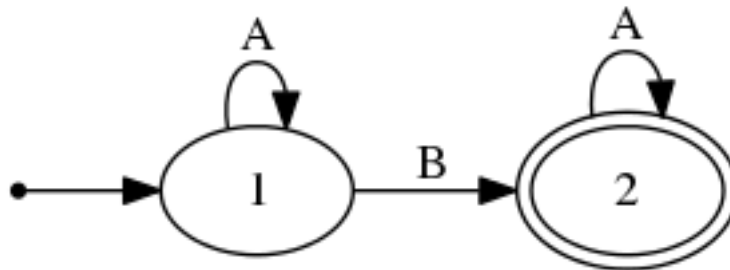

Figure 3.3. Automata which reads a*ba*

3.3.3. Informal behavior

An automaton is used to decide whether a word — a sequence of symbols — is part of a language (the word is accepted by the automaton) or not (the word is rejected). An automaton gets one input at a time. When run, an automaton has a current state which is the initial state at the beginning. Then, at each step, on input symbol is received. If there is no transition labelled with this symbol outgoing from the current state, then the word is rejected. If it exists such a transition, the symbol is accepted and the current state of the automaton becomes the state targeted by the transition. A work is accepted if all its symbols have been accepted and the last current state of the automaton is an accept state. Otherwise the work is rejected.

An automaton is nondeterministic if it contains a transition with no symbol (it may be fired as soon as its source state is the current state) or if it contains two transitions with the same source state and the same symbol. An automaton is either deterministic or nondeterministic.

3.3.4. Scenarios

The scenarios related to automata shared the same structure as they all consist in checking whether a work is accepted or rejected by an automaton. It thus consists in feeding the automaton with letters (symbols) of the work (from the first to the last one) and then to indicate that the end of the work has been reached.

If we consider the word `aba`, the input scenario is :

1. Feed symbol `a'

2. Feed symbol `b'

3. Feed symbol `a'

4. Terminate

The automaton will then answer. The response can be 'accepted' or 'refused'.

3.3.5. Examples of models

We give here some examples of automaton with the language they model and some examples of accepted and rejected words.

Deterministic automata

We first consider some simple example of automata for which there is only one outgoing transition for a state.
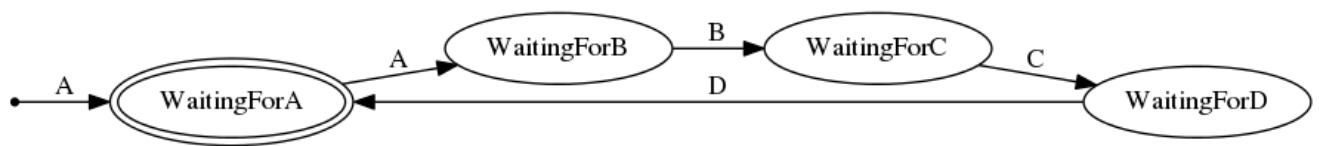

Figure 3.4. Automata which reads (ABCD)*

- Examples of accepted words: (empty word), ABCD, ABCDABCD, etc.

- Examples of rejected words: ABC, ABCDA, D, etc.

TODO: Other examples:

- only one state

- several states and only one final state

- several outgoings transition for one state

- several final states.

Dictionary Automata:
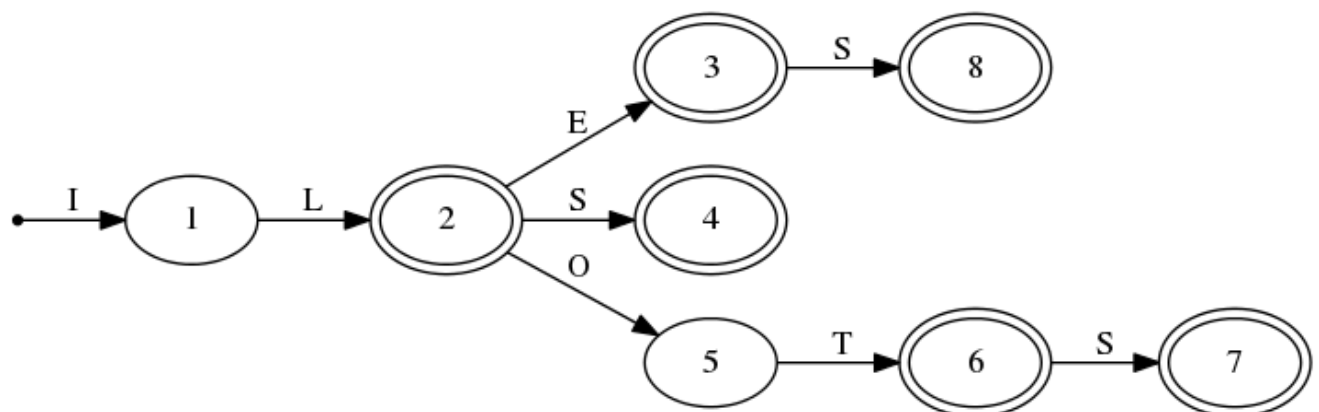
Figure 3.5. Automata which reads words

The only accepted words are: IL, ILE, ILES, ILS, ILOT, ILOTS

TODO: To be translated in English.

Nondeterministic automata

TODO: Several transitions with the same symbol.

TODO: A transition with no associated symbol (automatic transition?)
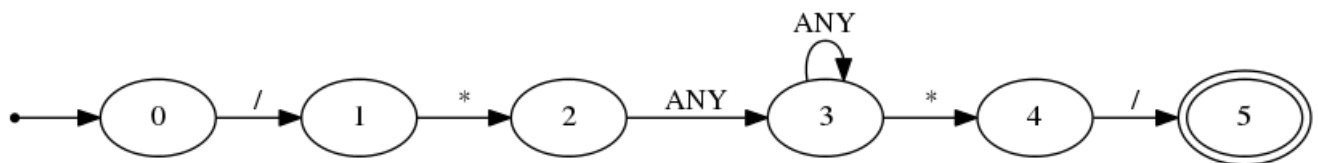
Figure 3.6. Automata which reads C commentary

Pushdown automata

For example to check that open and close symbols are well suited.

3.4. Creating an xDSML Project

First, start by creating a new xDSML project (*New > Project > GEMOC Project / new xDSML Project*), with your desired name (for instance "com.example.automata"). In the created project, we can open the project.xdsml file. The xDSML view summarizes all the important resources used in an xDSML project (which are part of and managed by other projects). This view is a kind of dashboard or control center to have quick access to any important resource of the project.

3.5. Increment 1 : Deterministic Automata

3.5.1. Specification of the xDSML interface

In this step, we describe the interface of the language. It includes interface to the system engineers (for example AS and CS) but also to other models and xDSML (AS, DSE, EF and ED).

3.5.2. Define the Abstract Syntax (AS)

To define the AS we can either select an existing project (Browse button) or create a new one. To create a new one, we click on "EMF project" on the xDSML view of project.xdsml. Let us call it "org.example.automata.model". Let us call our package "automata". We will use the default ns URI and ns Prefix. We may then edit the Ecore MetaModel either with the graphical editor or with the tree editor.
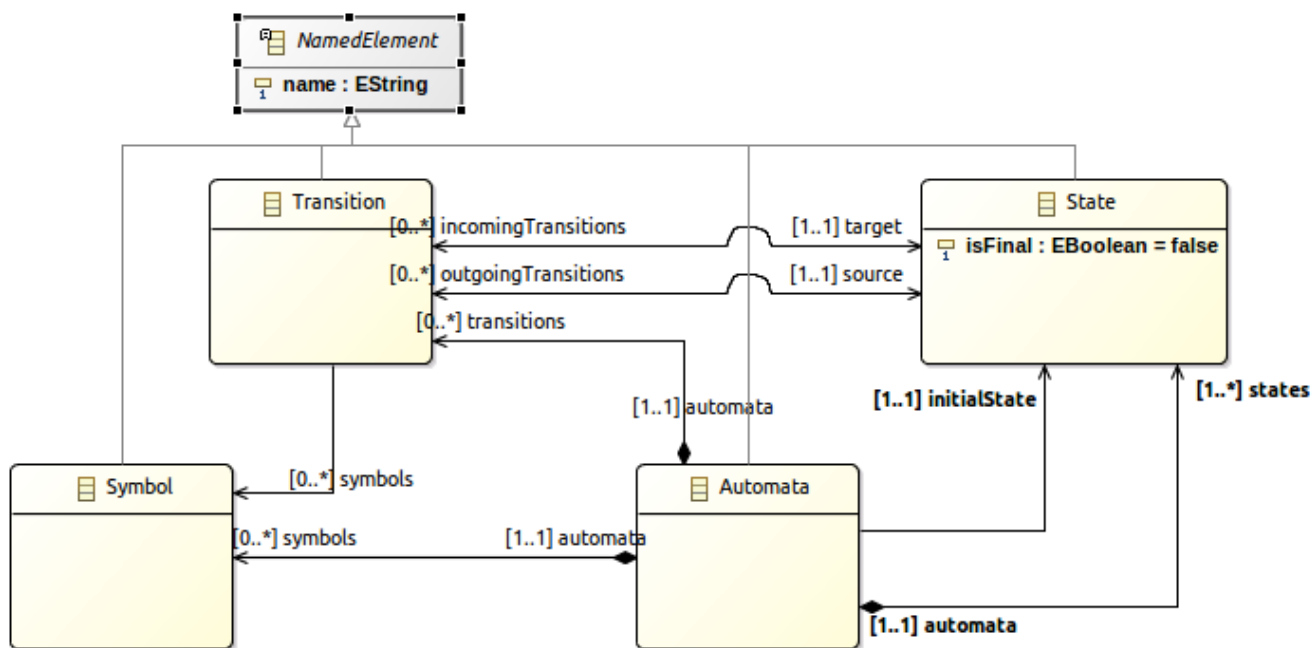
Figure 3.7. Automata Metamodel

An Automaton is a composed of States (at least one), Transitions and Symbols. An automaton has an initial state (reference). A state can be a final state (attribute). A Transition must have a source and a target, both of type State. A Transition is fired upon occurrence of one of its associated Symbol. For practical reasons, we also add EOpposite references whenever possible. Therefore States, Transitions and Symbols know which Automata they belong to. Symbols know which Transition(s) they are referenced by. States know their outgoing and incoming Transitions. Automatas, States, Transitions and Symbols all have a name (factorized in the NamedElement metaclass).

Once the Ecore MetaModel is done, we can come back to the xDSML view. The "EMF project" and the "Genmodel URI" have been updated.

Set the "Root container model element" to "automata::Automata".

Open the associated Genmodel (click on Genmodel URI) to generate the Model Code, Edit Code and Editor Code by right clicking on the root of the Genmodel (right-click on root element). The packages "automata", "automata.impl" and "automata.util" as well as the plug-ins "com.example.automata.model.edit" and "com.example.automata.model.editor" are generated.

3.5.3. Define concrete syntaxes (CS)

A concrete syntax is convenient way to view or edit a model. It can be textual (Xtext project for example) or graphical (Sirius project for example). They can be added to the xDSML project like we have done for AS.

For now, we postpone the design of the Concrete Syntaxes until we are sure the semantics has been correctly implemented.

> ⚠️ **Warning**
>
> Therefore, a graphical Concrete Syntax is required in order to use the graphical animator later on during simulations.

3.5.4. Identifying DSE

Domain Specific Events are part of the interface of the language and allow communication with the system engineer and the other models of the system.

For our Automata xDSML, we decide that there are 3 events which are of relevant interest to the environment (user through a GUI or another xDSML through language composition operators):

Initializing the automata
    occurs only once at the start of the simulation

Injecting a symbol
    occurs when the user gives a new symbol of the work to test

Terminating the automata
    occurs when the user has given all the symbols of the word. It is used to indicate the end on the word.

> 💡 **Tip**
>
> Other DSE may be of interest, for example firing a transition, rejecting a symbol, etc. They would be output events (the already identified ones being input events).

At this moment, DSE are defined in an ECL (Event Constraint Language) file. In the xDSML view, click on *ECL Project* to create a DSE Project. Let us name it "com.example.automata.dse" (it is the proposed name). In the corresponding field, place the path to the Ecore MetaModel ("platform:/resource/com.example.automata.model/model/automata.ecore") and make sure the "Root container model element" is "automata::Automata" and name the file "automataDSE". Ignore the error that is displayed.

Right click on the DSE project and make sure that in "configure", the "DSE builder" functionality is active.

An error is indicated in the newly created project. To correct it, fill-in the "moc2as.properties" file by completing the property with the name of the root element. In our case, that is "rootElement = Automata".

For now, we will complete the ECL file with the following elements:

- **Metamodel import:** (already initialized) Domain-Specific Events and MoCC constraints are defined in the context of a concept from the AS, so the first thing we need is to import the metamodel.

```
import 'platform:/resource/com.example.automata.model/model/automata.ecore'
```

- **Domain-Specific Events specification:** here we can define MoccEvents and a mapping towards EOperations present in the Metamodel (XXX). The first step is to identify which behaviors should be schedulable by the MoCC, and which should be seen as part of the behavioral interface of the xDSML.

Therefore, we define three Domain-Specific Events by defining three MoccEvents each referencing an Execution Function (implemented later).

```
package automata
 context Automata
  def: mocc_initialize : Event = self.initialize()
  def: mocc_terminate : Event = self.terminate()

 context Symbol
  def: mocc_occur : Event = self.occur()
endpackage
```

TODO: Write DSE without mapping them to DSA.

> **⚠ Warning**
>
> The signature of the Execution Functions needs to be present in the MetaModel. Therefore, we need to modify the Ecore MetaModel and add the three following operations:
>
> - Automata.initialize()
>
> - Automata.terminate()
>
> - Symbol.occur()
>
> To represent methods with Void as return type in EMF, do not complete the field "EType" of the EOperations.

> **💡 Tip**
>
> If the AS is changed (automata.ecore), we have to do "Reload…" on the genmodel, generate again the Model, Edit and Editor, and re-register the ecore. Nevertheless,

> the ECL is not always able to see the changes. In such a case close the editor and
> open it again. It should work.

3.5.5. Defining Domain-Specific Actions (DSA)

DSA includes the definition of Execution Data (ED) and Execution Functions (EF). They are both implemented in Kermeta 3 in 'K3 Aspect project' whose lastname is, by convention, 'k3dsa'.

Click on *K3 project* in the xDSML view (Behavioral definition / DSA definition). The wizard to create of new Kermeta 3 project is launched with the name of the project initialized (k3dsa is the last name).

Default options can be kept except for the value of *Use a template based on ecore file* field which must be changed from *None* to *Aspect class from ecore file*.

We can now finish the wizard.

Clicking again on *K3 project* will now allow to choose and open automata.xtend. It has been initialized with a template that can be discarded.

We can now complete the Kermeta 3 file (automata.xtend) with the definition of ED and EF.

Execution Data (ED)

We identify two runtime information for Automata. The first one stores the current state of the automaton. It is called 'currentState', a reference to State. Its value is either the one of the state of the automaton or the 'null' value. The 'null' value indicates that a symbol has not been accepted by the automaton.

The second ED stores the status of the symbols being analysed, either accepted or rejected. It is modelled as the 'accepted' boolean.

TODO: Define a new class in DSA ErrorState which extends State? When in the error state, the automate rejects every symbols.

To add 'currentState' and 'accepted' execution data, we define them in an Aspect on the Automata class as follow.

```
@Aspect(className=Automata)
class AutomataAspect {
    public State currentState
    public boolean accepted
}
```

> **Warning**
>
> If you plan to use the Graphical animation, then comment the code above and add this reference to the Ecore Metamodel directly. This is due to how the animator connects to the Abstract Syntax (for now).

Execution Functions (EFs)

Here are the execution functions we decide to define. The three first functions corresponds to the DSE already identified, the other ones are Helpers which ease the writing of the code of the previous ones. For each of this operation a logging is done.

Automata.initialize()

initialize the automaton: set its current state to its initial state and accepted to true.

Automata.terminate()

decide whether the sequence of symbols has been accepted or rejected by the automaton.

Symbol.occur()

makes the automaton read a new occurrence of this symbol. It is the main execution functions. It relies on the following helper functions.

State.getTransitions(Symbol s)

returns the list of all the outgoing transitions of this state which accept the s symbol. It is a **Query** execution function.

Automata.read(Symbol s)

This automaton reads the symbol s. It updates the current state according to the possible outgoing transitions of the current state and the symbol s. If there is only one possible transition, its target state becomes the new current state (delegated to Transition.fire() helper). If there is several possible transitions, then the automaton is Nondeterministic and an exception is raised. Finally, is there is no possible transition, the current state becomes an error ('currentSate' is set to 'null') the state and the sequence of symbols will be rejected. If the automaton was already in an error state, then nothing happens.

Transition.fire()

change the current state of the automata: the new state is target state of this transition. A precondition checks that the source state of the transition is the current state of the automata. An exception is thrown if the precondition fails.

Here is the complete 'automata.xtend' file with the code of all execution functions (and execution data).

automata.xtend.

```
package automata
```

```
import java.util.logging.Level
import java.util.logging.Logger

import static extension automata.AutomataAspect.*
import static extension automata.SymbolAspect.*
import static extension automata.StateAspect.*
import static extension automata.AutomataAspect.*
import static extension automata.TransitionAspect.*

import fr.inria.diverse.k3.al.annotationprocessor.Aspect

@Aspect(className=Automata)
class AutomataAspect {
 static private Logger logger = Logger.getLogger(typeof(Automata).getName())

 public State currentState
 public boolean accepted

 def public void initialize() {
  _self.currentState = _self.initialState;
  _self.accepted = false;
  _self.logger.info("[" + _self.name + "] Initialized, currentState is " +
 _self.currentState.name + ".")
 }

 def public void terminate() {
  _self.logger.info("[" + _self.name + "]" + "Finished.")
  // XXX: ne marche pas
  _self.accepted = _self.currentState != null && _self.currentState.isFinal
  val cs = _self.states.filter[ it === _self.currentState ]
  _self.accepted = cs.size > 0 && cs.head.isFinal
  var result = "rejected" // XXX better way to write it?
  if (_self.accepted) {
   result = "accepted"
  }
  // throw new RuntimeException("Finished. Word is " + result)
   // throwing an exception is the only way for the moment to
   // force the simulation to end.
 }

 //@ Helper with arguments
 def void read(Symbol s) {
  _self.logger.info("[" + _self.name + "]" + "read(" + s.name + ").")
  if (_self.currentState == null) {
   _self.logger.finer("** Already in the error state!");
  } else {
   val possibleTransitions = _self.currentState.getTransitions(s)
   val size = possibleTransitions.size
   // FIXME: I have not been able to write it with a switch :(
```

```
    if (size == 0) { // No possible transition
     _self.logger.finer("No transition for symbol " + s.name + " from state "
 + _self.currentState.name)
     _self.currentState = null
     _self.accepted = false // useful?
    } else if (size == 1) { // only one possible transition
     var singleTransition = possibleTransitions.head
     _self.logger.finer("Only one possible transition: " +
 singleTransition.name)
     singleTransition.fire()
    } else { // nondeterministic
     throw new RuntimeException("Non deterministic automaton: "
       + "several transitions accept symbol " + s.name
       + " in state " + _self.currentState.name)
    }
   }
  }

  def String toString() {
   // XXX To be improved
   var String str = "States : "
   str += _self.states.map[ s | (if (s == _self.currentState)  '[' + s.name +
 ']' else s.name)
    + (if (s.isFinal) '!' else '')]
   str
  }

}


@Aspect(className=State)
class StateAspect {

 //@ Helper (Query) with arguments
 def package Iterable<Transition> getTransitions(Symbol s){
  _self.outgoingTransitions.filter[symbols.filter[name == s.name].size > 0]
 }

}


@Aspect(className=Symbol)
class SymbolAspect {
 static private Logger logger = Logger.getLogger(typeof(Symbol).getName())

 def public void occur() {
  _self.logger.info("[" + _self.automata.name + "]" + "Symbol " + _self.name
 + " occurred.")
  _self.automata.read(_self) // call an helper DSA
```

```
   }

 }


@Aspect(className=Transition)
class TransitionAspect {
 static private Logger logger =
 Logger.getLogger(typeof(Transition).getName())

 def package void fire() {
  Contract.require(_self.automata.currentState === _self.source,
    "[" + _self.automata.name + "]" + "Source state of " + _self.name
    + " (" + _self.source.name + ") is not the current state (" +
 _self.automata.currentState.name + ")")
  _self.logger.info("[" + _self.automata.name + "]" + "Fired Transition " +
 _self.name + ".")
  _self.automata.currentState = _self.target
 }

 }
```

Testing DSA

Once the DSA are written --- or, even better, while they are written --- we must test them.

First, we define a method that feeds an automaton with a word (each letter of the word is considered as a symbol). It is defined in the AutomataExecution class. Here is the xtend code.

```
package automata

import static extension automata.AutomataAspect.*
import static extension automata.SymbolAspect.*

class AutomataExecution {

    def static boolean accepted(Automata a, String word) {
        println("==== What about " + word + "?")
        a.initialize()
        for (var i = 0; i < word.length(); i++) {
            val c = word.charAt(i)
            val ss = a.symbols.filter[name.equals("" + c)]
            if (ss.size == 0) { // unknown symbol for the automaton
                return false
            } else {
                ss.get(0).occur()
            }
        }
        a.terminate()
        println("==== What about " + word + "? : " + a.accepted)
        a.accepted
    }
```

```
}
```

Then, we can write a classical JUnit TestCase that tests an automaton on some tests using the 'AutomataExecution.accepted(String word)' method. Here is an example of such a test case.

```java
package automata;

import org.junit.Test;
import static org.junit.Assert.*;
import static automata.AutomataIO.*;
import org.eclipse.emf.ecore.resource.Resource;

public class AutomataTest {

    @Test
    public void testerAStar() {
        Resource model = loadResource("../org.example.automata.as/model/aS.xmi");   // XXX
        Automata a = (Automata) model.getContents().get(0);

        assertTrue(AutomataExecution.accepted(a, "a"));
        assertTrue(AutomataExecution.accepted(a, "aaaaa"));
        assertTrue(AutomataExecution.accepted(a, ""));
        assertFalse(AutomataExecution.accepted(a, "aaabaaa"));
        assertFalse(AutomataExecution.accepted(a, "c"));
    }

}
```

3.5.6. Model of Concurrency and Communication (MoCC)

At the moment the MoCC is defined using MoCCML, a superset of CCSL. It is composed of two parts. The first one is the reusable one, defined in a MoCCML project. It contains the declaration and definition of relations on clocks (MoCC events). At the top level, a state machine can be used to specify the relations. The second one explains how to use those relations according to the abstract syntax of the DSML. Is part of the DSE/ECL project.

> **i** **Note**
>
> We will use the term clock as a synonym of MoCC event, that is events that are managed by the MoCC. The purpose is to avoid confusion with domain specific events (DSE).

We have already identify DSE and we have defined them in the ECL file (DSE project). Implicitly, these DSE events are mapped to corresponding clock at the MoCC level. So, we have three clocks, initialize, occur and terminate.

We want that the 'initialize' clock clicks only once and before all other clocks. Then we can have any occurrences of the 'occur' clock and, eventually, one occurrence of the 'terminate' clock.

TODO: It could be defined using a state machine : initialize, then occur *, then terminate.

Reusable part of the MoCC: MoCCML

Here, we want that 'initialize()' DSE occurs only once before any other event. Thus we define a **relation** called 'FirstAndOnlyOnce' whose purpose is specify that a first clock will happen only once, before all the others clocks. Thus, it takes two arguments, the first clock, the collection of other clock. Its prototype is as follows:

```
RelationDeclaration FirstAndOnlyOnce(first : clock, other : clock)
```

Then, we have to provide the definition **(RelationDefinition)** which satisfies the this specification. The mocclib file hereafter provides both the declaration and the definition of this relation.

automata.mocclib.

```
StateRelationBasedLibrary automataLib{
 imports{
  import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/
ccsllibrary/kernel.ccslLib" as kernel;
  import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/
ccsllibrary/CCSL.ccslLib" as ccsl;
 }

 RelationLibrary basicautomataRelations{
  RelationDeclaration FirstAndOnlyOnce(first : clock, other : clock)
  RelationDefinition FirstAndOnlyOnceImplem[FirstAndOnlyOnce]{
   Expression firstTickOfFirstEvent = OneTickAndNoMore(OneTickAndNoMoreClock
-> first)
   Expression firstTickOfOtherEvents =
OneTickAndNoMore(OneTickAndNoMoreClock -> other)
   Relation Precedes(
    LeftClock -> first,
    RightClock -> firstTickOfOtherEvents
   )
   Relation Coincides(
    Clock1 -> first,
    Clock2 -> firstTickOfFirstEvent
   )
  }
 }
}
```

First, we create a new MoCCML project (right click on the xDSML project > GEMOC Language > Create MoC Project) --- this action is not yet available from the xDSML view --- and place a library of custom MoCCML relations and expressions there. Let us call this project "com.example.automata.mocc.lib". We can now complete the automat.mocclib file.

Specific part of the MoCC

Next, we can define the actual constraints on the clock of an Automata model. It is described in the ECL file (SDE project) using AS concepts and the relations defined in the MoCCML project as well as the standard libraries relations. Thus, we start to import the lib and the ecore files (at top of the ECL file).

```
import 'platform:/resource/com.example.automata.as/model/automata.ecore'
ECLimport "platform:/resource/com.example.automata.mocc.lib/mocc/
automata.moccml"
ECLimport "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/
ccsllibrary/kernel.ccslLib"
ECLimport "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/
ccsllibrary/CCSL.ccslLib"
```

The main part of the ECL file specify how to instantiate clocks ('def' keyword and 'Expression' construction) and which constraints to put on them thanks to the 'Relation' concept.

First, we want to make sure that we do the initialization ('initialize' clock) of the Automata before anything else. Therefore, we will use the relation "FirstAndOnlyOnce" defined in our custom MoCCML library.

```
context Automata
 inv InitBeforeAnythingElse:
  let allOccurEvents : Event = Expression Union(self.symbols.mocc_occur) in
  let allOtherEvents : Event = Expression Union(allOccurEvents,
 self.mocc_terminate) in
  Relation FirstAndOnlyOnce(self.mocc_initialize, allOtherEvents)
```

Now, we also want to make sure that we can only inject one symbol at a time. This is modelled by a relation of exclusion between the MoccEvents corresponding to the injection of the symbols. Therefore we add the following constraint:

```
 inv ExclusivityOfSymbolOccurrences:
  Relation Exclusion(self.symbols.mocc_occur)
```

However we cannot both inject a symbol and terminate at the same time. Therefore we also need to add the following exclusion:

```
 inv ExclusivityOfSymbolsAndTerminate:
  let allSymbolOccurEvents : Event = Expression
  Union(self.symbols.mocc_occur) in
```

```
Relation Exclusion(self.mocc_terminate, allSymbolOccurEvents)
```

As soon as you save the ECL file, a .qvto file should be generated in the folders qvto-gen/ language and qvto-gen/modeling. Make sure that your xDSML project references the .qvto file that is available in qvto-gen/modeling.

> **Tip**
>
> If the Ecore file describing the syntax is changed while the ECL file is opened, it may be required to close the ECL file and to open it again to ensure modifications on the Ecore file are seen by ECL.

Testing the MoCC

TODO: To be completed

3.5.7. Using the Modeling Workbench

Technical Workarounds

A few workarounds are needed before you can launch the Modeling Workbench :

- TODO: Dans le projet xDSML, initialiser le champ Code executor class name with automata.xdsml.api.impl.AutomataCodeExecutor

- In the DSA Project, MANIFEST.MF, runtime, export the non-Java package containing your .xtend DSAs

- In the xDSML Project, plugin.xml, add the following attribute to the XDSML_Definition: modelLoader_class="org.gemoc.gemoc_modeling_workbench.core.DefaultModelLoader"

- In the xDSML Project, MANIFEST.MF, add the following dependency: org.gemoc.gemoc_modeling_workbench.ui, org.gemoc.gemoc_language_workbench.extensions.k3

- Make sure a .qvto has been generated in the your DSE Project /qvto-gen/modeling.

- TODO: Supprimer les import sur les aspects non utilises

- Dans project.xdsml, verifier que le QVT-o reference est celui du dossier qvto-gen/modeling du projet DSE.

- TODO

Testing and debugging the xDSML

Launch the Modeling Workbench. Create a new general project, for instance "com.example.automata.instances". In this project, create a new Automate instance (New > Other… > Automata Model) "ABCD.automata" whose root is of type Automata.

Create a Run Configuration: right click on the model and select "Run As… > Run Configurations". Create a new "Gemoc eXecutable Model" configuration. Model to execute: "/com.example.automata.instances/ABCD.automata", xDSML: "automata". Change the "Decider" to "Step by step user decider".

> **Warning**
>
> In "Animator" place any valid .aird. This issue should be solved in the next iteration of the Studio. In the panel "Common", select "Shared file" and put the project path there: "/com.example.automata.instances". Give a name to the configuration like "Automata ABCD".

3.6. Increment 2: new MoCC and DSA for Automata (MoCC focused version)

In the previous version the MoCC is only responsible of ensuring that DSE events (input symbols and end of word event) arrive one at a time. The DSA choose the right transition to fire, if any. To do so, fire() has been considered as an helper. It can thus be called from the occur() DSA of Symbol element.

We now propose another solution that gives more responsibilities to the MoCC: it will decide which transition can be fired.

Principle: Transition.fire() in no more an Helper but a Modifier that will be scheduled by the MoCC. We add clock on state to know whether a state is current or not a simulation step. To be defined.

TODO: to be developed.

3.7. Increment 3: Graphical visualization

- On the model (current states, the symbols already analyzed?)

- A tabular presentation ?

- A specific view as a Gantt

TODO: to be developed.

3.8. Increment 4: Consider nondeterministic automata.

TODO: to be developed.

- Two transitions with the same symbol and the same source state.

- A transition with no label.

Principle: currentState become currentStates. We maintain the set of all states that are accessible by the symbols already accepte by the automaton.

3.9. Increment 5: Pushdown automaton

TODO: The purpose of Pushdown Automaton is to illustrate the Feedback mechanism.

We will now extend our automaton to include a stack. The feedback mechanism will be used to decide whether a transition is firable according to the symbol on top of the stack.

> **Note**
>
> These aspect will be included in a future version of the tutorial when the proposed approach to handle feedbacks will have been integrated to the GEMOC studio.

3.10. Increment 6: Call of user actions

> **Note**
>
> Will be added in a future version of this tutorial.

3.11. Increment: TBD

TODO: Which other increments?

3.12. TODO

- Definition of AS should be part of this tutorial (correct NOTE: at the beginning).

- Pourquoi automataDSE ? (DSE/ECL part)

- Define concrete Syntax with Sirius. Sould it be done in the second Eclipse?

- rename mocc_* to dse_* to reflect the fact that we would like them to be DSE. We must explain in the mapping DSE/MoCC that at this moment, each DSE generates on MOCCEvent.

- rename xDSML.model to xDSML.as?

- Expliquer les automates en partant des exemples, pour eviter la redondance avec l'explication du MM et plus logique dans l'optique de du system engineer.

- Process: ensure a better conformance of the text with the process

- AS: accept state could be represented as an attribute of the State element (instead of a reference)

- Examples have to be redone to make it more clear, smaller (graphic representation), etc.

- Define a style for the block GEMOC, use the GEMOC logo.

- Static semantics? To be mentioned. We could provide the OCL constraint which check whether the model will be executable (non indeterminism).

- Give examples of models, **scenarios** and **expected results**.

- Add expected results in term of animation : want to see current states, executable transitions, incoming symbols…

- A way to formalize scenario (including expected results)?

- MetaModel or Metamodel or AS?

- K3: Can we have several @Aspect(className=Automata) class AutomataAspect { }, one for the ED, one the EF?

Forme du tutoriel :

- Define a style to highlight text from xtend, ecl, moccml, etc.

- Definir un nouveau type de block GEMOC avec le Logo GEMOC.

- Comment definir l'equivalent de --atribute tabsize=4 dans le .asciidoc directement ?

To be more efficient: * Directly include source code rather than pasting it in this doc.

To be improved:

- DSE part: for the moment ECL has several purpose. It is thus confusing. We must first focus on the definition of the DSE and not on the various mappings.

To be added:

- Explain somewhere: When the .ecore is changed. The genmodel must be updated and the code generated again.

Studio improvements:

- Could the moc2as.properties file in DSE/ECL project be automatically initialized (from the Root container model element)?

MoCCML:

- How to represent a collection of clocks in a RelationDeclaration rule?

Tutorial focus:

- Illustration of the process

- Demonstrative in term of GEMOC studio basic operations

Tutorial maintenance:

- Which version of GEMOC Studio to use?

- What frequency for updating the tutorial?

- How contribute to the tutorial after the first version?

- Keep a single file or split the tutorial in several mini-tutorial?

- Record a video? Once the tutorial has been validated !

Choices in the way of writing DSA/EF

- Use logging?

- Add a Contract class for preconditions?

- …

Bibliography

The bibliography lists some useful external documents. For a more complete list, please refer to the publications section on http://gemoc.org site.

Articles

[globalizing-modeling-languages]   Globalizing   Modeling   Languages   [http://hal.inria.fr/ hal-00994551] (Benoit Combemale, Julien Deantoni, Benoit Baudry, Robert France, Jean-Marc Jezequel, Jeff Gray), In Computer, IEEE, 2014.

Glossary

AS
Abstract Syntax.

API
Application Programming Interface.

Behavioral Semantics
see Execution semantics.

CCSL
Clock-Constraint Specification Language.

Domain Engineer
user of the Modeling Workbench.

DSA
Domain-Specific Action.

DSE
Domain-Specific Event.

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.

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

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.

xDSML

Executable Domain-Specific Modeling Language.

Index

D

Decider, 6

E

Engine, 6, 7
Event Manager, 7, 7

G

GEMOC
   process, 16
      overview, 12

L

language designer, 1
Language workbench, 1
Logical Step, 6, 9

M

Modeling workbench, 5

O

overview, 12

P

process, 16
   overview, 12

R

Runtime Data, 7, 9

S

Sirius, 4, 4

T

TFSM, 1, 5
   Language workbench, 1
   Modeling workbench, 5
Timeline, 6, 7