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

Introduction vii

1. Language Workbench overview vii

2. Modeling workbench overview ix

3. General concerns and prerequisite ix

1. GEMOC Language Workbench 1

1.1. Overview 1

1.2. xDSML Project 2

1.2.1. Purpose 2

1.2.2. Creating the xDSML Project 2

1.2.3. Editing the xDSML Project 2

1.3. Domain Model Project 2

1.3.1. Purpose 2

1.3.2. Creating the Domain Model Project 2

1.3.3. Editing the Domain Model Project 3

1.4. Defining the Domain-Specific Actions (DSA) Project 3

1.4.1. Purpose 3

1.4.2. Creating the DSA Project 3

1.4.3. Editing the DSA Project 3

1.4.4. Testing the Domain-Specific Actions 4

1.5. Defining Domain-Specific Constraints 4

1.6. Defining a Concrete Syntax 4

1.6.1. Defining a Concrete Syntax with Xtext 4

1.6.2. Defining a Concrete Syntax with Sirius 4

1.7. Defining a Model of Concurrency and Communication (MoCC) 4

1.7.1. Introduction 4

1.7.2. The ECL approach to Identify DSE and constraints 5

1.7.3. The MoCCML approach to define constraints 6

1.8. Defining the Domain-Specific Events (DSE) 11

1.8.1. Purpose 11

1.8.2. Creating the DSE Project 11

1.8.3. Editing the DSE Project 11

1.9. Defining the Feedback Specification 12

1.9.1. Purpose 12

1.9.2. Creating a Feedback Policy 13

1.9.3. Semantics of the Feedback Policy 13

1.10. Defining a debug representation 14

1.10.1. The debug representation wizard 14

1.10.2. Implementation details 20

1.11. Process support view 22

2. Gemoc Modeling workbench 23

2.1. Modeling workbench overview 23

2.2. Editing model 24
2.2.1. Editing model with Sirius 24
2.2.2. Editing model with EMF Tree Editor 24
2.3. Executing model 24
2.3.1. Launch configuration 24
2.3.2. Engine View 26
2.3.3. Logical Steps View 26
2.3.4. Timeline View 26
2.3.5. Event Manager View 27
2.3.6. Animation View 27
2.3.7. Debug View 27
2.3.8. Variable View 27
3. GEMOC xDSML definition tutorial with Automaton DSML 28
3.1. Introduction 28
3.2. The GEMOC Approach for defining eXecutable DSML (xDSML) 29
3.2.1. Architecture of a GEMOC xDSML 29
3.2.2. Main Characteristics of the GEMOC Process. 31
3.2.3. Recommanded GEMOC Process 31
3.3. Definition of the requirements/expectations on the xDSML 34
3.3.1. Application Domain: Automata 34
3.3.2. Description of automata 35
3.3.3. Informal behavior 35
3.3.4. Scenarios 35
3.3.5. Examples of models 36
3.4. Creating an xDSML Project 37
3.5. Increment 1 : Deterministic Automata 37
3.5.1. Specification of the xDSML interface 37
3.5.2. Define the Abstract Syntax (AS) 37
3.5.3. Define concrete syntaxes (CS) 38
3.5.4. Identifying DSE 39
3.5.5. Defining Domain-Specific Actions (DSA) 41
3.5.6. Model of Concurrency and Communication (MoCC) 46
3.5.7. Using the Modeling Workbench 49
3.6. Increment 2: new MoCC and DSA for Automata (using State Machines of MoCCML) 50
3.7. Increment 3: new MoCC and DSA for Automata (MoCC focused version) 50
3.8. Increment 4: Graphical visualization 50
3.9. Increment 5: Consider nondeterministic automata. 51
3.10. Increment 6: Pushdown automaton 51
3.11. Increment 7: Call of user actions 51
3.12. Increment: TBD 51
3.13. TODO 51
Bibliography 54
Glossary 55

Index 57

List of Figures

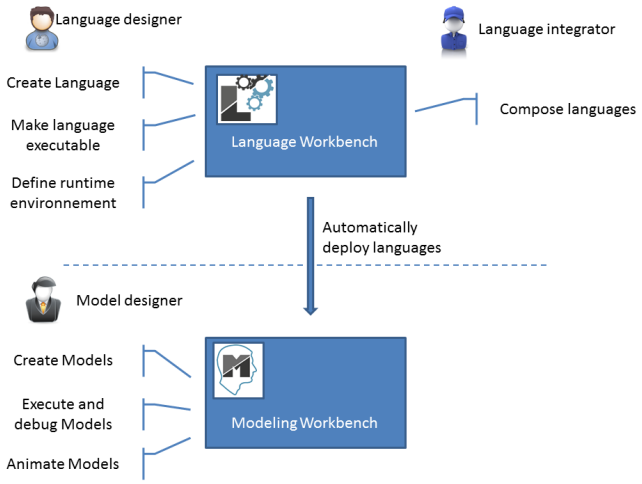
1.1. Screenshot of the GEMOC Language Workbench showing the design of a Timed Finite State Machine (TFSM) example. 1
1.2. Screenshot of the First graphical level of Edition in MoCCML. 8
1.3. Screenshot of the Second graphical level of Edition in MoCCML (Constraint Implementation). 10
2.1. Screenshot of GEMOC Studio Modeling Workbench on the TFSM example (execution and animation). 23
3.1. Overview of the GEMOC process 30
3.2. Recommended GEMOC Process 33
3.3. Automata which reads a*ba* 35
3.4. Automata which reads (ABCD)* 36
3.5. Automata which reads words 36
3.6. Automata which reads C commentary 37
3.7. Automata Metamodel 38

Introduction

The GEMOC Studio offers 2 main usages:

- Building and composing new executable DSML. This mode is intended to be used by language designers and language integragtors (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*.



In this document:

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

1. Language Workbench overview

The GEMOC Language Workbench allows building and composing new executable DSMLs.

To acheive that, it offers appropriate tools and guidance for the underlying activities wich are grouped in categories.

Build executable language. The first main activity is to build an executable language. The language designer is assited by a dedicated Dashboard to guide him during his work.

This first dashboard groups the activities as follow:

Create a language
: This activity can be decomposed in activities in two groups :

  Domain model (abstract syntax)

  - Create the abstract syntax with EMF (incl., the ecore model and the genmodel)

  Model editors (concrete syntax)

  - Create a graphical editor with Sirius Designer
  - Create a textual editor with Xtext

Make your language executable
: This activity is decomposed in 2 main parts:

  Sequential Execution semantics

  - Create the execution semantics with Kermeta and Melange

  Concurrent Execution semantics

  - Create the Domain-Specific Actions with Kermeta and Melange
  - Create the Domain-Specific Events with ECL
  - Create a library of Model of Computation with MoccML

Define the runtime environnement for the language

  Model Debugger

  - Select your breakpointable elements

  Model dynamic representation (for debugging, animation, monitoring)

  - Create the graphical representation with Sirius Animator
  - Plug your own add-ons

TODO add links to the corresponding sections in the main chapters

TODO: add snapshot of the dashboard

Compose executable languages. The second main activity is to define the composition between executable languages. The language integrator is also assisted by a dedicated dashboard that proposes the following activity groups:

Compose languages
: This activity can be decomposed in activities in two groups :

Create composition operators

- Create operators with BCOoL

Combine languages

- Select a set of language to combine
- Apply composition operators to a set of languages.
- Customize composition

TODO: add snapshot of the dashboard

2. Modeling workbench overview

The GEMOC Modeling Workbench allows the use of available languages to create and execute models.

Create Models

- Use available editors to create domain models.

Execute and debug models

- Use execution engine
- Use step by step debugger
- Analyse time trace

Animate models

- Use animation views

3. General concerns and prerequisite

Note

In order to run, Eclipse workbench work better with additional memory. Use the following setting to start Eclipse: -Xms1024m -Xmx1024m -XX:PermSize=512m -XX:MaxPermSize=512m

Chapter 1. GEMOC Language Workbench

1.1. Overview

The **GEMOC Language Workbench** is intended to be used by the Language Designers and the Language Integrators. It provides the tools to create and compose eXecutable Domain-Specific Modeling Languages (xDSMLs) using the GEMOC approach.

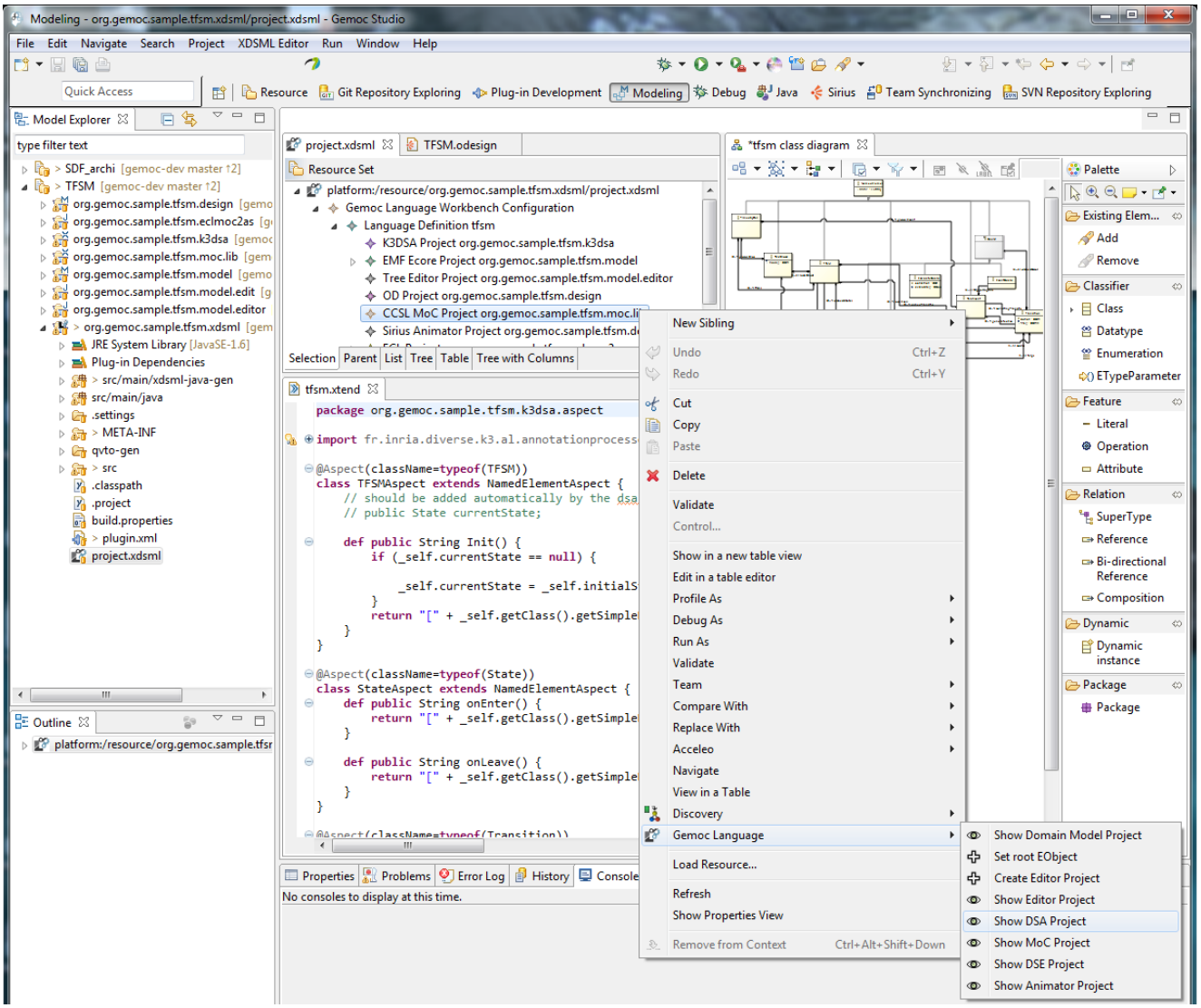


Figure 1.1. Screenshot of the GEMOC Language Workbench showing the design of a Timed Finite State Machine (TFSM) example.

1.2. xDSML Project

1.2.1. Purpose

The xDSML Project is the core project of languages created using the GEMOC Studio. It has two main purposes:

- Referencing the different projects implementing the various parts constituting an xDSML;
- Guiding the Language Designer in the language creation process through the GEMOC Dashboard illustrating the workflow of creating an xDSML using the GEMOC Studio.

1.2.2. Creating the xDSML Project

In the GEMOC Studio, go to: *File* > *New* > *Project…* > *New GEMOC Language Project*. This will create a new xDSML Project in your workspace.

1.2.3. Editing the xDSML Project

The main file constituting the xDSML Project is the *project.xdsml*. To edit this file, open it with the *XDSML Model Editor*.

From the information contained in this file, the project generates additional code that is used by the Execution Engine during the execution of models conforming to the xDSML (see Chapter 2, *Gemoc Modeling workbench*).

1.3. Domain Model Project

1.3.1. Purpose

The Domain Model Project specifies the concepts of the domain at hand and the structural relations between the concepts.

1.3.2. Creating the Domain Model Project

The GEMOC Studio relies on the Eclipse Modeling Framework for its Domain Model Projects. See the EMF website [http://eclipse.org/modeling/emf/] for more information on how to create an EMF project in Eclipse. The Domain Model is materialized as an Ecore metamodel.

When your EMF Project is done, connect your xDSML Project to it by specifying in the *project.xdsml* file the name of the EMF Project, the path to the genmodel of your Ecore metamodel and the name of the root element as *package::root*.

TODO: Melange?

1.3.3. Editing the Domain Model Project

If you wish to modify your Domain Model, do not forget to reload the associated genmodel and regenerate the EMF model code (and edit/editor code if you use them).

1.4. Defining the Domain-Specific Actions (DSA) Project

1.4.1. Purpose

The Domain-Specific Actions define the runtime state (**Execution Data**) of the model and the operations (**Execution Functions**) which modify the runtime state of the model.

1.4.2. Creating the DSA Project

In the GEMOC Studio, the DSA are implemented using Kermeta 3 [https://github.com/diverse-project/k3/wiki]. To create a new DSA Project, in the main menu of the GEMOC Studio, go to: *File > New > Project… > K3 Project*. In the wizard, create it as a Plug-in with EMF using the template of your choice. Then, connect the xDSML Project to the DSA Project by referencing the DSA Project in the *project.xdsml* file.

1.4.3. Editing the DSA Project

Kermeta 3 is based on xTend [http://www.eclipse.org/xtend/index.html]. The Execution Data and Execution Functions are defined through aspects weaved onto the metaclasses of the Domain Model.

Defining the Execution Data

The Execution Data consist in attributes and references added to existing concepts (metaclasses) of the Abstract Syntax. They may also include new metaclasses which define the type of these new attributes and references.

Defining the Execution Functions

The Execution Functions define how the Execution Data evolve during the execution of the model. Execution Functions can be implemented by defining the body of a method.

Note

For now, Execution Functions are considered as **atomic, instantaneous and blocking**. This means that any long computation will block the rest of the simulation, and concurrent Execution Functions are not executed in concurrence yet.

Warning

For technical reasons, the Domain Model (Ecore metamodel) must specify the signature of the Execution Functions as EOperations.

1.4.4. Testing the Domain-Specific Actions

It is possible to test the DSA (in particular the Execution Functions) by simply writing a simple program with a *main* function (using Java or Xtend/Kermeta3). Create or load a model conform to your Domain Model and call the Execution Functions in the right order to verify there are no runtime exceptions or domain issues.

1.5. Defining Domain-Specific Constraints

TODO

1.6. Defining a Concrete Syntax

An xDSML can support different concrete syntaxes. Most EMF-based editors should work with GEMOC, however the GEMOC Studio provides additional support for some specific editors. Thus, we recommend using: an EMF arborescent editor, an Xtext editor, and/or a Sirius editor. Don't forget to link the xDSML Project to the concrete syntax project(s) you want to use by editing the *project.xdsml* file.

1.6.1. Defining a Concrete Syntax with Xtext

See the Xtext website [http://www.eclipse.org/Xtext/].

1.6.2. Defining a Concrete Syntax with Sirius

If you want to create a graphical concrete syntax you can use Sirius. The Sirius documentation [http://www.eclipse.org/sirius/doc/] provides information for Sirius Specifier Manual [http://www.eclipse.org/sirius/doc/specifier/Sirius%20Specifier%20Manual.html].

1.7. Defining a Model of Concurrency and Communication (MoCC)

TODO: MoccML + ECL

1.7.1. Introduction

In the GEMOC approach, the executability characterization for a given language is done through several steps that include: the description of the actions associated with the language concepts (i.e. the Executions Functions); the description of the data / attributes that capture the state of a

model or its evolution (the Execution Data); the description of the underlying language model of concurrency (the MoCC constraints).

A Model of Concurrency and Communication (MoCC) represents the concurrency, synchronizations and the possibly timed causalities in the behavioral semantics of a language. It must represent the acceptable schedules of the atomic actions of the language, which represent both computation and communication.

In this part of the guide, we assume that achieving the first two points has already been done based on ???. In this part, we will mainly focus on the steps for the description and integration fo the model of concurrency in the language, once the EF and ED have been already taken into account and integrated to the language. The execution engine of the GEMOC Studio is based on an event-based semantics, which means that events are used to activate the EF actions that can change the state of a model.

To describe and integrate the model of concurrency, we must 1) have a description of the useful events allowing activation of the EF, 2) a mechanism to express the MoCC constraints that apply on these events.

- 1) is realized using the **ECL** language whose description and use is provided below
- 2) Is achieved through the **MoCCML** language whose description and use is described below

1.7.2. The ECL approach to Identify DSE and constraints

Overview of ECL

TODO

Creation of an ECL model GEMOC Studio

ECL models are created via files with the extension _.ecl. The xDSML environment enables the creation and / or opening of new ECL project that gives access to several functionalities such as (model transformation .qvto generator, etc) link to ???

How to use ECL

In this section, we show an exemplified way to use ECL. The required steps to go from a given language to the integration of its concurrency model are presented.

Import Language

As shown in Listing ???, an ECL file imports the extended language that contains the language metaclasses, the EF and ED. The ECL language defines the notion of **import** to import the extended metamodel (see import of dplExtended.ecore in Listing ???). The import is used to load all the concepts of language that are used to clarify: the EF activation events; static properties numeric or Boolean values, and constraints associated with concepts.

```
import 'platform:/resource/org.gemoc.model.dpl.dplextended/model/dplExtended.ecore'
```

Define Event-Action Creation

The ECL model gives access to the concept of the language via the declaration of their package container. In the illustration below, the package container is 'dplextended'. Every metaclass/ concept is declared as a 'context'. In a given context, we define the mapping between events and actions and the MoCC constraints of the context. The illustration shows the definition and mapping of events associated with the concept of 'Philosopher'. For the context 'Philosopher', the events are used to enable its actions ie eat() and think() actions. ECL syntax for declaring the mapping Event / Action is as follows:

- Define the type of the event: def: think: Event
  - When the event activate actions: the action associated to the event is accessed through the context via 'self.actionName()' (e.g. in Listing ???'Philosopher' self.think() and self.eat())
  - When the event does not trigger an action i.e. used as internal control events : Its declaration does not mention any related action (e.g. getrightFork, getleftFork, putrightFork, putleftFork)
- Finally, we can define local attributes to capture the value of static properties that can be for instance integers or booleans (e.g. def: eatcycleMax: Integer = self.eatCycles) The above steps specify the declaration of events and attributes that are used to control the actions for each language concept. We can focus on the description of the model of concurrency defined using MoCCML.

```
package dplextended
 context Philosopher
  def: think : Event = self.think()
  def: getrightFork : Event = self
  def: getleftFork : Event = self
  def: eat : Event = self.eat()
  def: putrightFork : Event = self
  def: putleftFork : Event = self
  def: thinkcycleMax : Integer = self.thinkCycles
  def: eatcycleMax : Integer = self.eatCycles
endpackage
```

The latter phase is done in two steps: the implementation of the execution control constraints with MoCCML; the use of these constraints in the context definition to specify how the events should be scheduled (determine their causalities).

1.7.3. The MoCCML approach to define constraints

This section presents the MoCCML editor that supports the edition of MoCCs. To keep the de#ned models formal and to provide a solver for the MoCC, an operational semantics and a solver based

on this formal semantics were defined. For more information on the operational semantics the reader can refer to MoCCML Operational Semantics [https://hal.inria.fr/hal-01060601v1].

Creating a MoCCML model in the GEMOC Studio

The MoCCML models are created via files with an _.moccml extension. They are also natively created from the dashboard xDSML, where you can create a MoCC project. The project defines an empty model _.moccml with just the name of the library to be created. For a graphical representation of the models, you have to right-click on the file (New Representation File) and run next until the creation of the _.aird representation. This procedure is described in the Sirius tutorial Sirius documentation [http://www.eclipse.org/sirius/doc/] and Sirius Specifier Manual [http://www.eclipse.org/sirius/doc/specifier/Sirius%20Specifier%20Manual.html] to create new diagrams starting from a model whose graphical editor was made from Sirius.

Overview de MoCCML

MoCCML is a declarative meta-language specifying constraints between the events of a MoCC. At any moment during a run, an event that does not violate the constraints can occur. The constraints are grouped in libraries that specify MoCC speci#c constraints. The constraints are eventually instantiated to de#ne the execution model of a speci#c model. The execution model is a symbolic representation of all the acceptable schedules for a particular model. MoCCML is based on the principle of defining constraints on events. There are two categories of constraint de#nitions: the Declarative De#nitions and the Constraint Automata De#nitions. Each constraint definition has an associated ConstraintDeclaration that defines the prototype of the constraint.

Presentation of the MoCCML Editor

The concrete syntax of MoCCML is implemented as a combination of graphical and textual syntaxes to provide the most appropriate representation for each part of a MoCC model library. The graphical syntax can be divided into two levels of representation: one for the definition of the MoCC libraries (the declaration and definition of the automata constraints); another for the implementation of the constraints in the form of automata. For instance:

- The first level of representation contains elements as illustrated in Figure Figure 1.2, “Screenshot of the First graphical level of Edition in MoCCML.”. The represented model imports two CCSL libraries (kernel.ccslLib and CCSL.ccslLib). The imported libraries provide predefined types that are used to define formal parameters such as DiscreteClocks, Integers, etc. Each defined Relation Declaration is associated to a Automata constraint definition. The association is done through the ‘Set Declaration Relation’ link.

- The second level of graphical representation defines the graphical syntax for the modeling of the Automata constraints.

- The overall MoCC models are serialized to a textual syntax, which means that the graphical models are transformed into their equivalent representation in a textual formal. **Both**

representations (graphical or textual) can be used for edition of models. Moreover, we define the integration of an embedded textual editor in the graphical representation to focus on specific parts of the MoCC model that are better edited using a textual syntax (eg trigger, the guards and the actions on transitions). Embedded editors are called by double-clic, and are placed on specific graphical edition elements (Relation Declaration, Relation Definition, DeclarationBlock, Transition).

Example-Driven use of MoCCML

NB: MoCCML has multiple pallets to instantiate a library. The pallets are located on the right branch of the editor. The creation of new library is preceded by an import of the native CCSL libraries (kernel.ccslLib, CCSL.ccslLib) which provide primitives for the description of events and variables that are handled by the constraints in the MoCC library. We use the third pallet in Figure Figure 1.2, “Screenshot of the First graphical level of Edition in MoCCML.”to import such CCSL libraries.

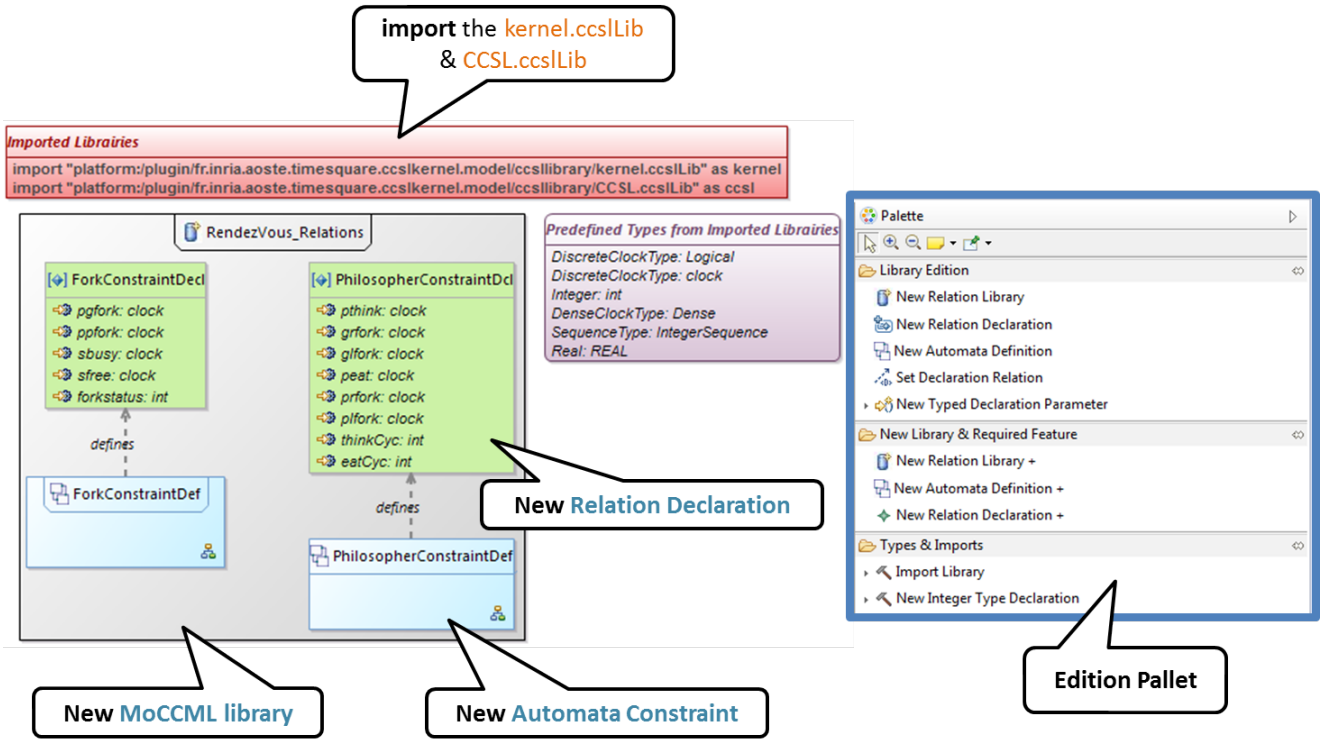


Figure 1.2. Screenshot of the First graphical level of Edition in MoCCML.

Creating MoCC Libraries

As shown in Figure Figure 1.2, “Screenshot of the First graphical level of Edition in MoCCML.”, creating new MoCC libraries can be done by using the first two pallets on the right (Library Edition, New Library & Required Feature). In these pallets, the element (Library New Library + New Relationship and Relationship) can be used for the instantiation of a new MoCC library. The two are distinguished by the fact that the last mentioned will create a new library of MOCC, while

adding a default Relation Declaration. In Figure Figure 1.2, “Screenshot of the First graphical level of Edition in MoCCML.” we create a new Library called 'RendezVous_Relations'.

Declaring the constrained events

In a MoCC library, we define constraints and their declarations. The declarations identify events and parameters to be considered in the implementation of the constraint. In the editor, the declaration is made using the two above mentioned pallets, and using the elements in the pallets i.e.: 'New Relationship Declaration' and 'New Relationship Declaration +'. The two differ in that the latter creates a Relation Declaration with a default formal parameter declaration. In the Figure Figure 1.2, “Screenshot of the First graphical level of Edition in MoCCML.”, we create two relation declarations (ForkConstraintDecl and PhilosopherConstraintDecl). Listing ??? also shows the equivalent textual code generated for the PhilosopherConstraintDecl.

```
RelationDeclaration PhilosopherConstraintDcl(
   pthink : clock,
   grfork : clock,
   glfork : clock,
   peat : clock,
   prfork : clock,
   plfork : clock,
   thinkCyc:int,
   eatCyc:int
  )
```

Defining the constraints

The implementation of constraints can be specified textually or graphically. Graphically, the first two pallets are used to create new definitions of constraints associated with their declarations. Constraint definitions is done using the menu items (New Automata Definition and New Automata Definition +). In Figure Figure 1.2, “Screenshot of the First graphical level of Edition in MoCCML.”, the following constraints are specified: ForkConstraintDef, PhilosopherConstraintDef). At this stage, we toured the main notions that can be set on the first level of graphical description with MoCCML. To navigate in the second level of graphical description (Constraint implementation), one should right-click on a specified constraint definition using (Open Diagram / New Diagram). Open Diagram will navigate to an existing diagram; New Diagram will create a new diagram to edit. The MoCCML Editor offers 3 different pallets for: editing the automata, defining the local variables and editing the transitons (ie adding Trigger, Guard, Actions). Figure Figure 1.3, “Screenshot of the Second graphical level of Edition in MoCCML (Constraint Implementation).” shows a simple example with two control states. An additional Layer displays the details of the transitions (Trigger, Guard, Action) as shown in Figure Figure 1.3, “Screenshot of the Second graphical level of Edition in MoCCML (Constraint Implementation).”, see yellow boxes. Besides, editing DeclarationBlock boxes and 'details in Transitions' can be done using embedded text editor by double-clicking on the related boxes. We can then edit the properties of transitions and local variables textually.

One can define the desired set of constraints on the concepts of language using the MoCCML editor. To see the text code corresponding to the serialization of the edited MoCC models, the user

can open the _.moccml file. Editing can also be directly made from this file and all the changes will be reflected in the graphical editor. The use of constraints is shown in the next section.

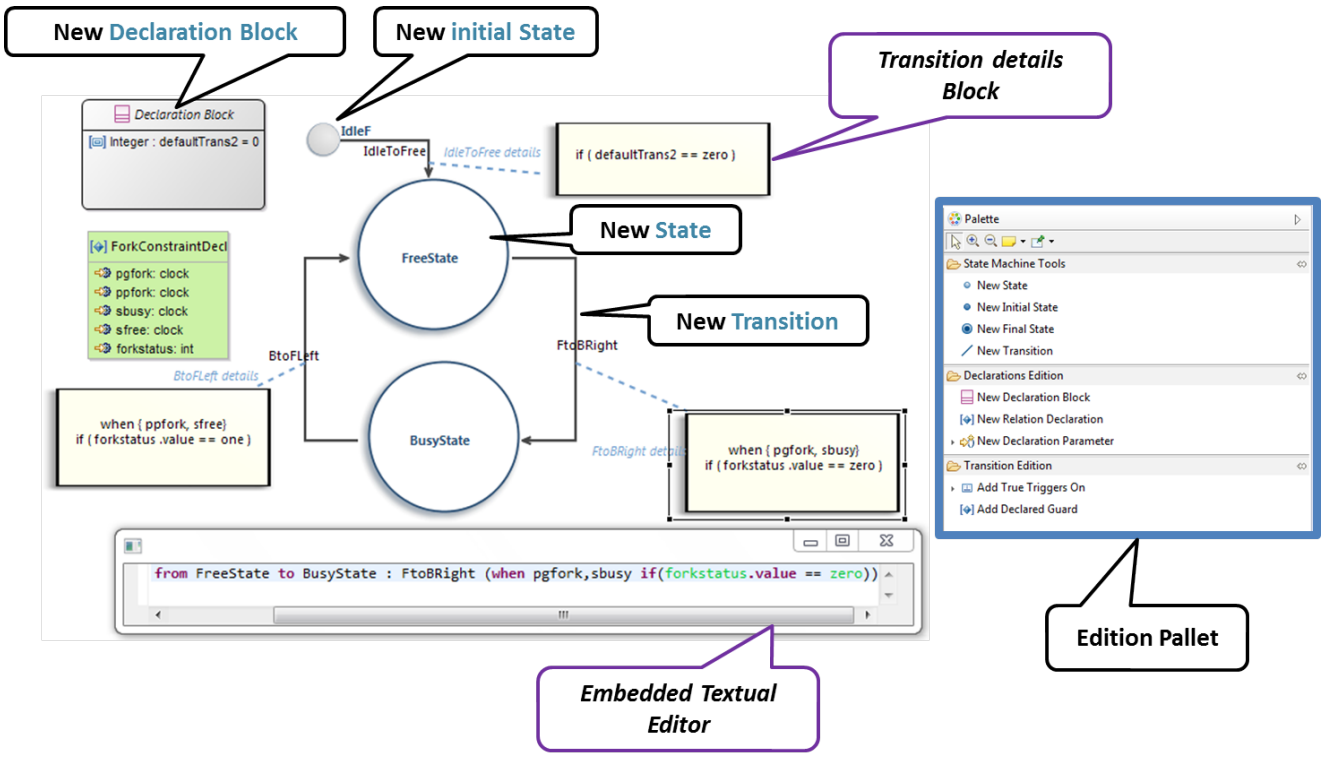


Figure 1.3. Screenshot of the Second graphical level of Edition in MoCCML (Constraint Implementation).

Using the constraints on the ECL

The MoCC constraints models can be used in the ECL file on concepts which they are attached. To declare the constraint on the events, we re-declare the context of the concept then define an invariant 'inv', see Listing ???. In this listing we also import the MoCCML library that was defined previously (i.e. rendez_vous.moccml) For instance, the invariant related to the context Philosopher (PhilosopherConstraintInv) uses the PhilosopherConstraintDef via its PhilosopherConstraintDcl declaration? It takes as input the set of control events and static variables used to calculate the causality between events.

```
ECLimport "platform:/resource/org.gemoc.dpl.xdsml.mocc.model/mocc/rendez_vous.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"

package dplextended
 context Philosopher
  def: think : Event = self.think()
  def: getrightFork : Event = self
  def: getleftFork : Event = self
```

```
  def: eat : Event = self.eat()
  def: putrightFork : Event = self
  def: putleftFork : Event = self
  def: thinkcycleMax : Integer = self.thinkCycles
  def: eatcycleMax : Integer = self.eatCycles

 context Philosopher
 inv PhilosopherConstraintInv:
  Relation PhilosopherConstraintDcl(
                self.think,
                self.getrightFork,
                self.getleftFork,self.eat,
                self.putrightFork,
                self.putleftFork,
                self.thinkcycleMax,
                self.eatcycleMax
                )
endpackage
```

1.8. Defining the Domain-Specific Events (DSE)

1.8.1. Purpose

The DSE define a mapping between MoccEvents from the MoCC and the Execution Functions. They specify the coordination between what happens in the MoCC, depending on its constraints, and which operations must be called in the model (thus resulting in changes in the Execution Data). They are also exposed as the behavioral interface of the xDSML, based on which the Behavioral COmposition Operator Language (BCOoL) defines the composition of xDSMLs.

1.8.2. Creating the DSE Project

Create an empty Plug-in Project. In this project, create a new file with the ".GEL" extension. The DSE can be defined using the **Gemoc Events Language** (GEL).

1.8.3. Editing the DSE Project

The first step to designing the DSE in GEL consists in importing the MoCC2AS Mapping (ECL file) and the Domain Model (Ecore metamodel) of the xDSML. You can do so by using the *import platform:/resource/...* syntax at the beginning of the file.

Defining an Atomic Domain-Specific Event

Atomic Domain-Specific Events realize a 1-to-1 mapping between the MoccEvents and the Execution Functions of the xDSML. An atomic DSE is defined using the following syntax:

```
DSE <name>:
  upon <MoccEvent>
  triggers <Path-to-ExecutionFunction>
end
```

Where:

- <name> is the name of the Domain-Specific Event;
- <MoccEvent> is any MoccEvent from the MoCC2AS Mapping. This will also automatically define the DSE in the same context as the MoccEvent referenced, thus defining the starting point of navigation expressions from this DSE;
- <Path-to-ExecutionFunction> is a navigation path from the context of this DSE to an Execution Function.

Tip

In case of difficulty, make use of the auto-completion feature.

Note

The implicit context of a Domain-Specific Event is inferred from the context in which its associated MoccEvent is defined in the MoCC2AS Mapping. Therefore, the DSE will be usable for any instance in the model of the context of the DSE. For instance if a DSE is mapped to a MoccEvent defined in the context of the metaclass *Transition*, and there are 3 Transitions in the model being executed, this DSE will be usable for each of the 3 Transitions of the model.

Semantics of Atomic Domain-Specific Events

Upon an occurrence of the associated MoccEvent, an occurrence of the Domain-Specific Event is created and consumed by the GEMOC Execution Engine. Its consumption triggers the execution of the designated Execution Function.

Defining a Composite Domain-Specific Events

TODO

Semantics of Composite Domain-Specific Events

TODO

1.9. Defining the Feedback Specification

1.9.1. Purpose

Many languages need to be able to parameterize their control flow with their data flow. For instance, conditional statements like *if…then…else* or *switch…case* statements depend on the

result of a boolean expression. Since the GEMOC approach at designing xDSMLs promotes the separation of the concurrency (MoCC from the domain-specific data (DSA), there needs to be a form of communication from the DSA to the MoCC. For instance, the MoCC of an xDSML with a conditional statement specifies that after evaluating the condition, either the 'then' or the 'else' branch will be taken (in exclusion from one another), while the DSA provide the Execution Function that evaluates the condition, returning a boolean value. Making the link between the 'true' value and the 'then' branch, and between the 'false' value and the 'else' branch is the role of the Feedback Specification. More formally, the Feedback Specification specifies how domain data returned by an Execution Function must be interpreted to parameterize the MoCC. The Feedback Specification consists in a set of Feedback Policies which are associated to the atomic Domain-Specific Events. Therefore, the Feedback Specification is also done using GEL.

1.9.2. Creating a Feedback Policy

A Feedback Policy is associated to an atomic DSE by extending the syntax for defining the atomic DSE as follows:

```
DSE <name>:
  upon <MoccEvent>
  triggers <Path-to-ExecutionFunction> returning <resultName>
  feedback:
    [<filter1>] => allow <PathToMoccEventConsequence1>
    [<filter2>] => allow <PathToMoccEventConsequence2>
    ...
    default => allow <PathToMoccEventDefaultConsequence>
  end
end
```

Where:

- <resultName> is a local variable which will be affected with the value returned by the associated Execution Function;
- Between the *feedback* and *end* keywords, a set of Feedback Rules are defined (including a default Feedback Rule, with the *default* keyword). A Feedback Rule is composed of a filter (between squared brackets) and a consequence;
- <filterN> are predicates, most likely using <resultName> which define whether or not this Feedback Rule must be applied;
- <PathToMoccEventConsequenceN> is a navigation expression to a MoccEvent which specifies which MoccEvent is allowed as a result of the data returned by the Execution Function.

1.9.3. Semantics of the Feedback Policy

When the Execution Function associated to a DSE returns a result, the filter of every Feedback Policy is evaluated. If none of the filters is validated, then the default rule is applied. Else, the

collection of rules for which the filter validated are applied. For each rule to apply, the associated consequence is allowed to have occurrences in upcoming steps of execution. This is done by the Execution Engine by filtering out some solutions provided by the MoCC interpreter and only allowing the ones complying with the Feedback Specification.

Caution

Using the Feedback Specification implies the use of a certain design pattern in the MoCC. Make sure your MoCC is well-designed for using the Feedback Specification. Don't forget to check the tutorial concerning the Feedback Specification.

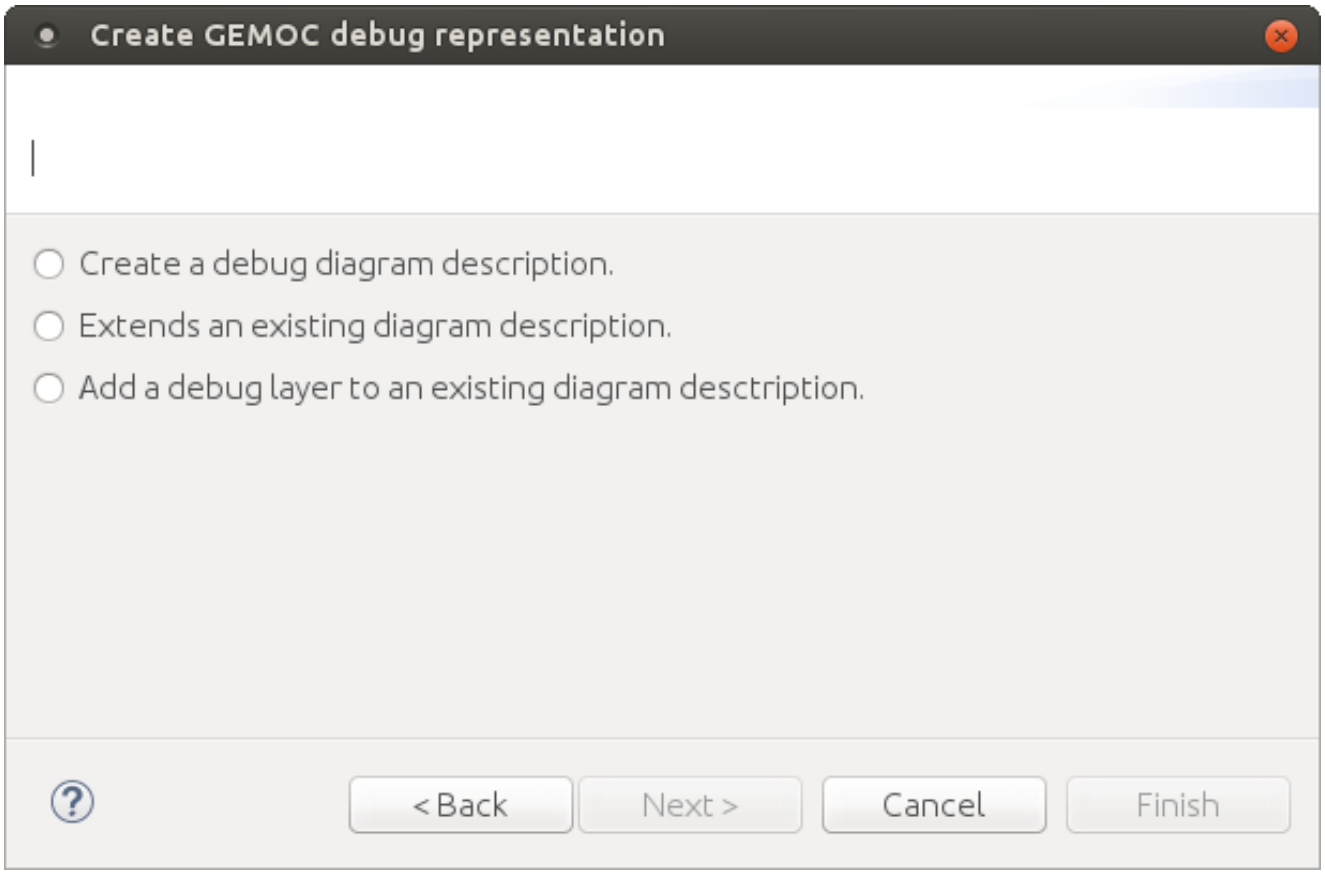
1.10. Defining a debug representation

The debug layer is an extension on top of a graphical editor defined with Sirius which represents runtime data and the current instruction. See Section 1.6.2, "Defining a Concrete Syntax with Sirius" for more details about Sirius. This section covers the debug representation creation wizard and the technical implementation details. Technical implementation details are only useful for advanced use case and troubleshooting.

1.10.1. The debug representation wizard

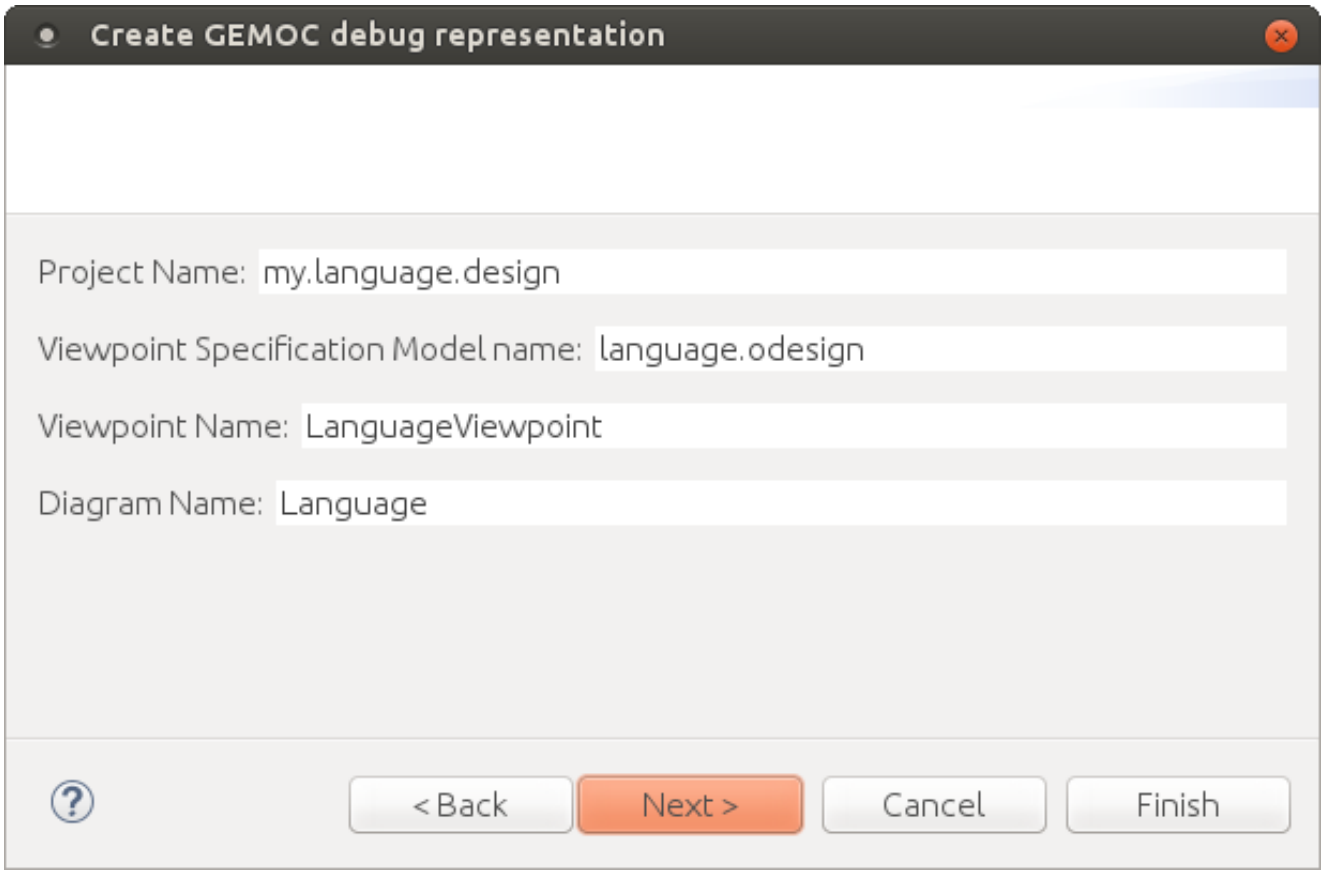
This wizard creates a layer to represent the current instruction and add commands in order to manage breakpoints and launch a simulation in debug mode. This is a default implementation, it can be customized to represent runtime data for instance. The customization uses the Sirius description definition, see the Sirius Specifier Manual [http://www.eclipse.org/sirius/doc/specifier/Sirius%20Specifier%20Manual.html] for more details. The wizard presents three ways to implement this layer:

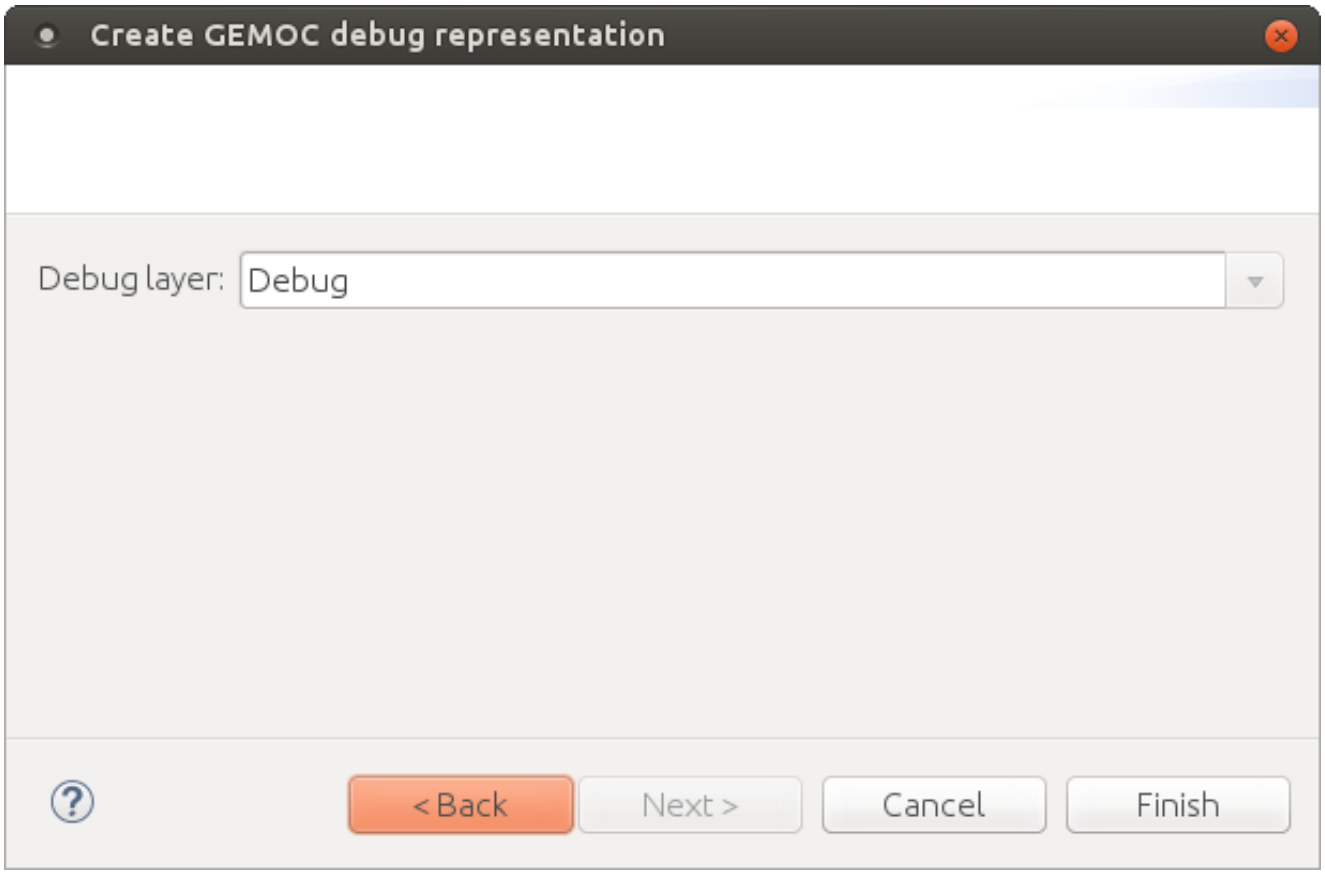
- Create a debug diagram description
- Extend an existing diagram description
- Add a debug layer to an existing diagram description



Create a debug diagram description

It creates a diagram representation with a default debug layer. The representation does not depend on another representation. A typical use case is a language where the runtime data representation is too far from the language graphical syntax.

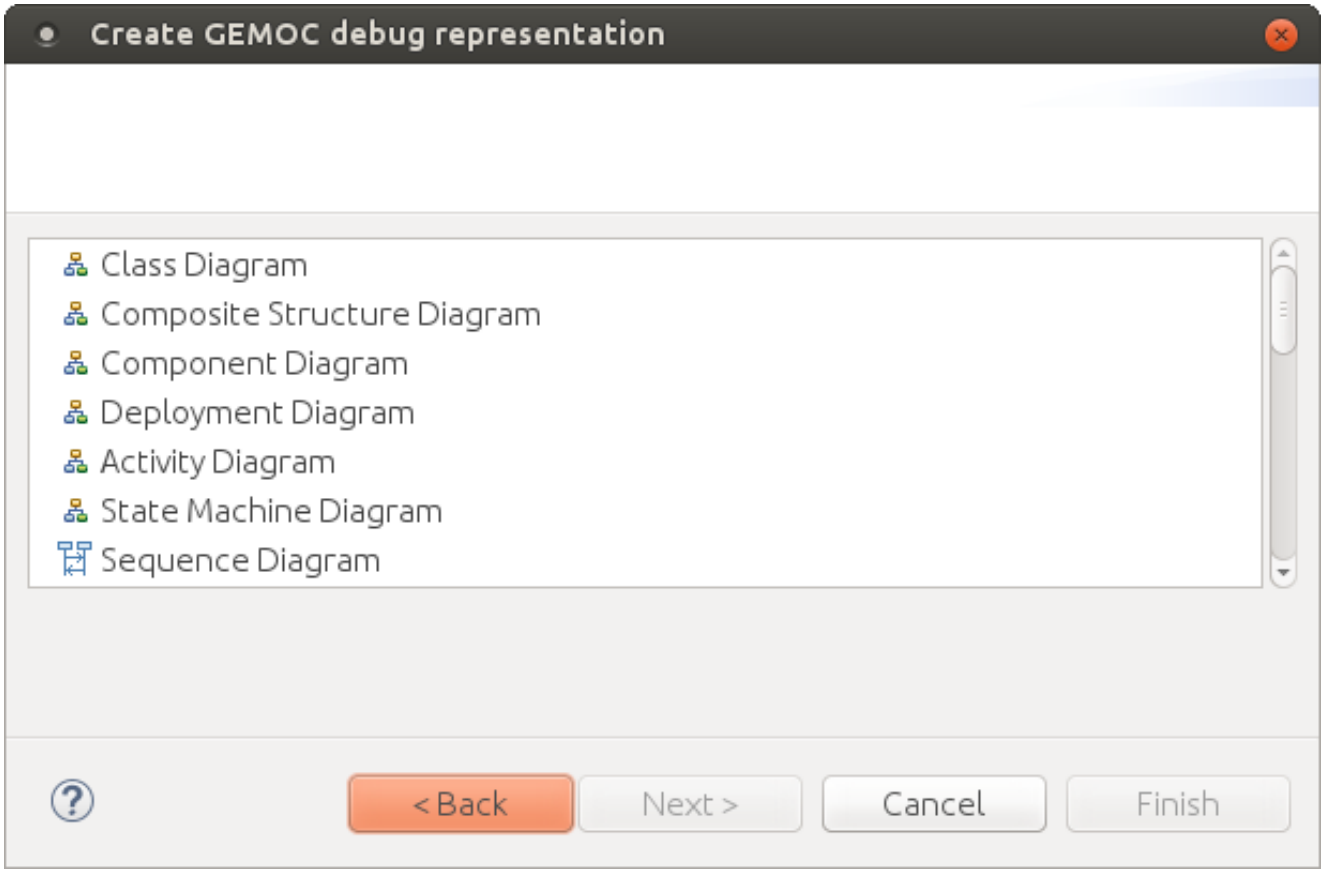

Create GEMOC debug representation
Project Name: my.language.design
Viewpoint Specification Model name: language.odesign
Viewpoint Name: LanguageViewpoint
Diagram Name: Language
< Back
Next >
Cancel
Finish


Create GEMOC debug representation
Debug layer: Debug
< Back
Next >
Cancel
Finish

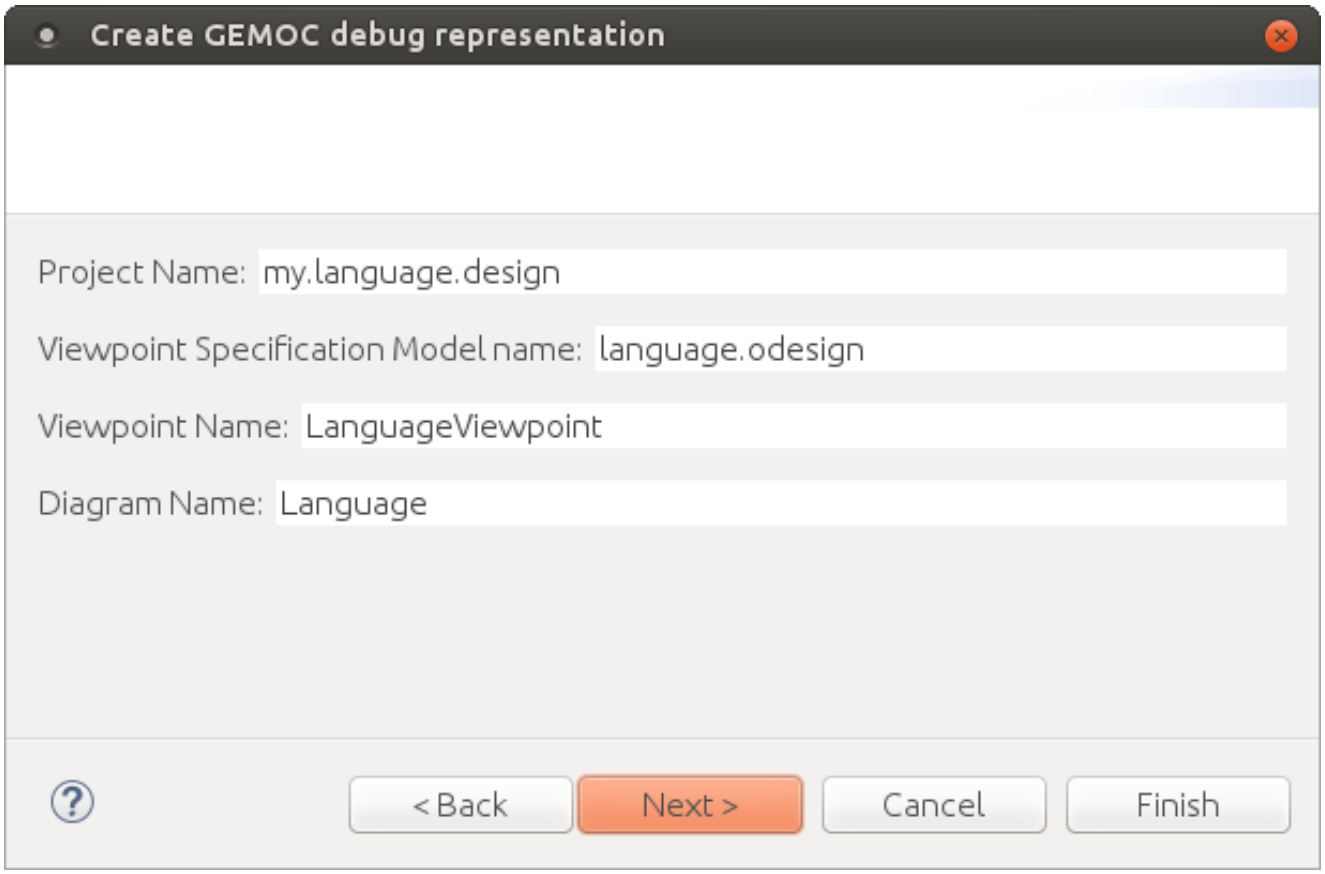
Extend an existing diagram description

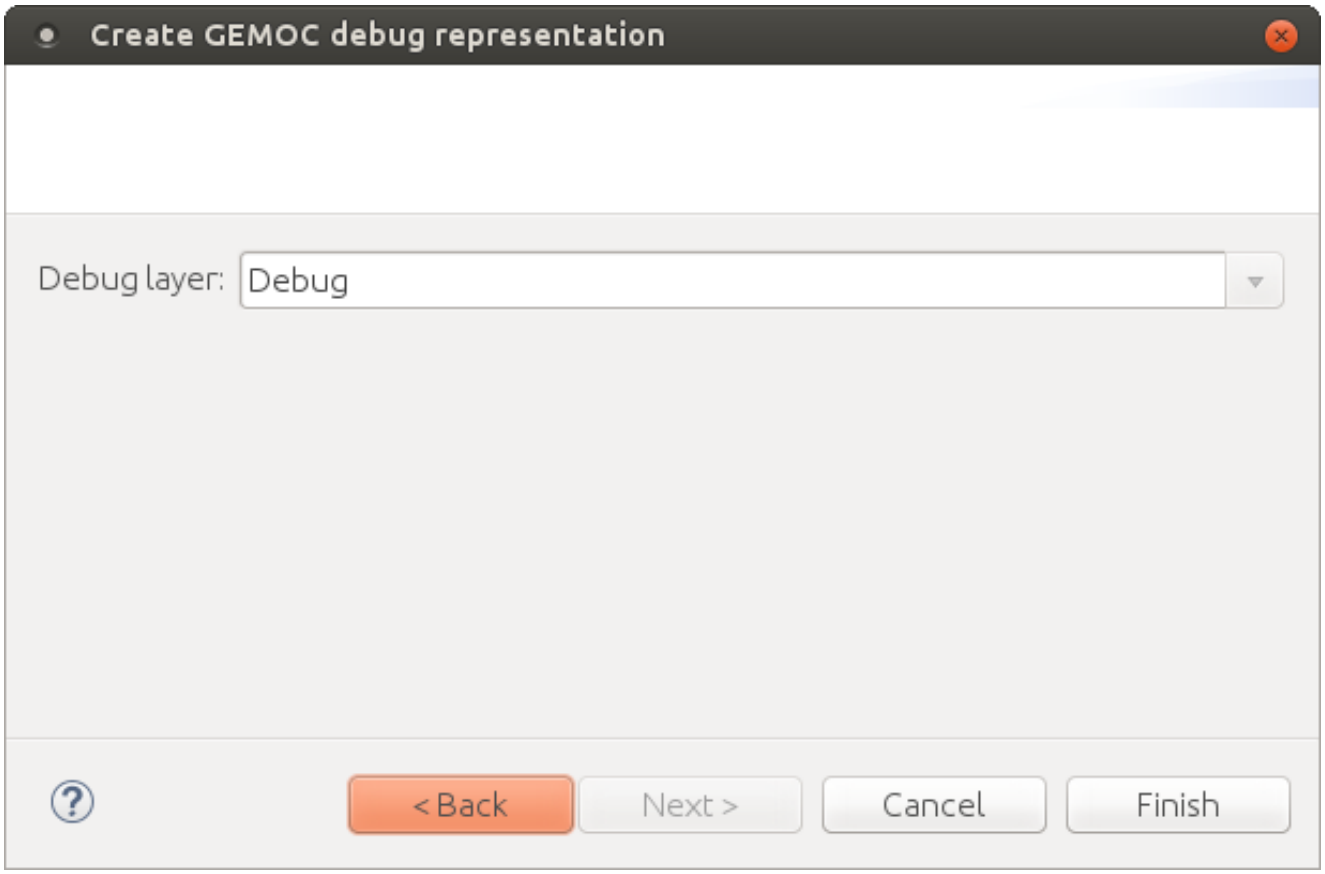
It creates a diagram representation with a default debug layer that extends an existing representation. This allows to have a debug layer based on the representation of the language concrete syntax. The language concrete syntax can be deployed without the debug representation. A typical use case is the reuse of an existing diagram definition that you cannot modify by yourself. For instance if you want to use UML [http://eclipse.org/modeling/mdt/?project=uml2], you can reuse the UML Designer [http://www.umldesigner.org/].

You can select any diagram description.



And then define the new diagram description which extends the one you previously selected.

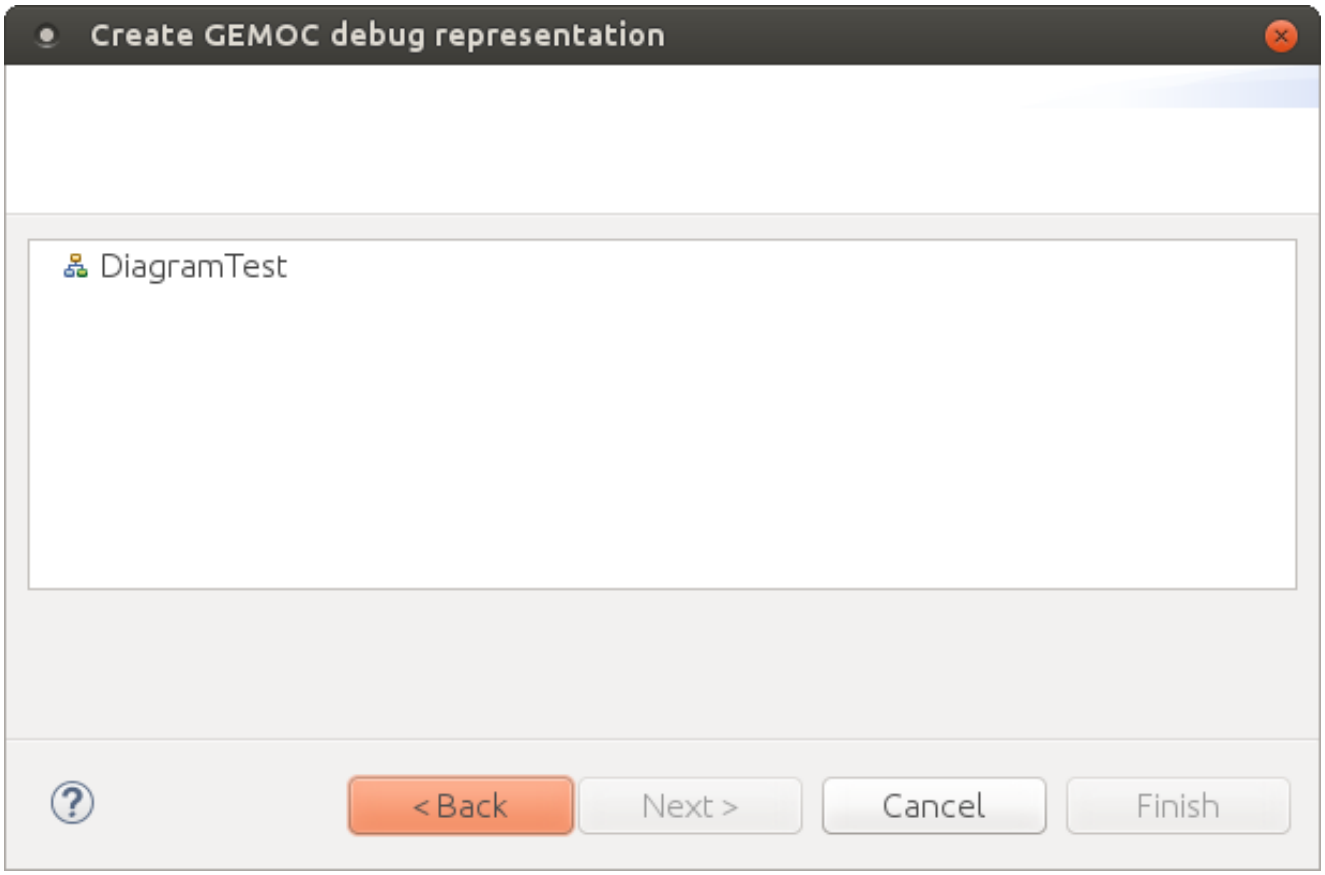

Create GEMOC debug representation
Project Name: my.language.design
Viewpoint Specification Model name: language.odesign
Viewpoint Name: LanguageViewpoint
Diagram Name: Language
< Back
Next >
Cancel
Finish

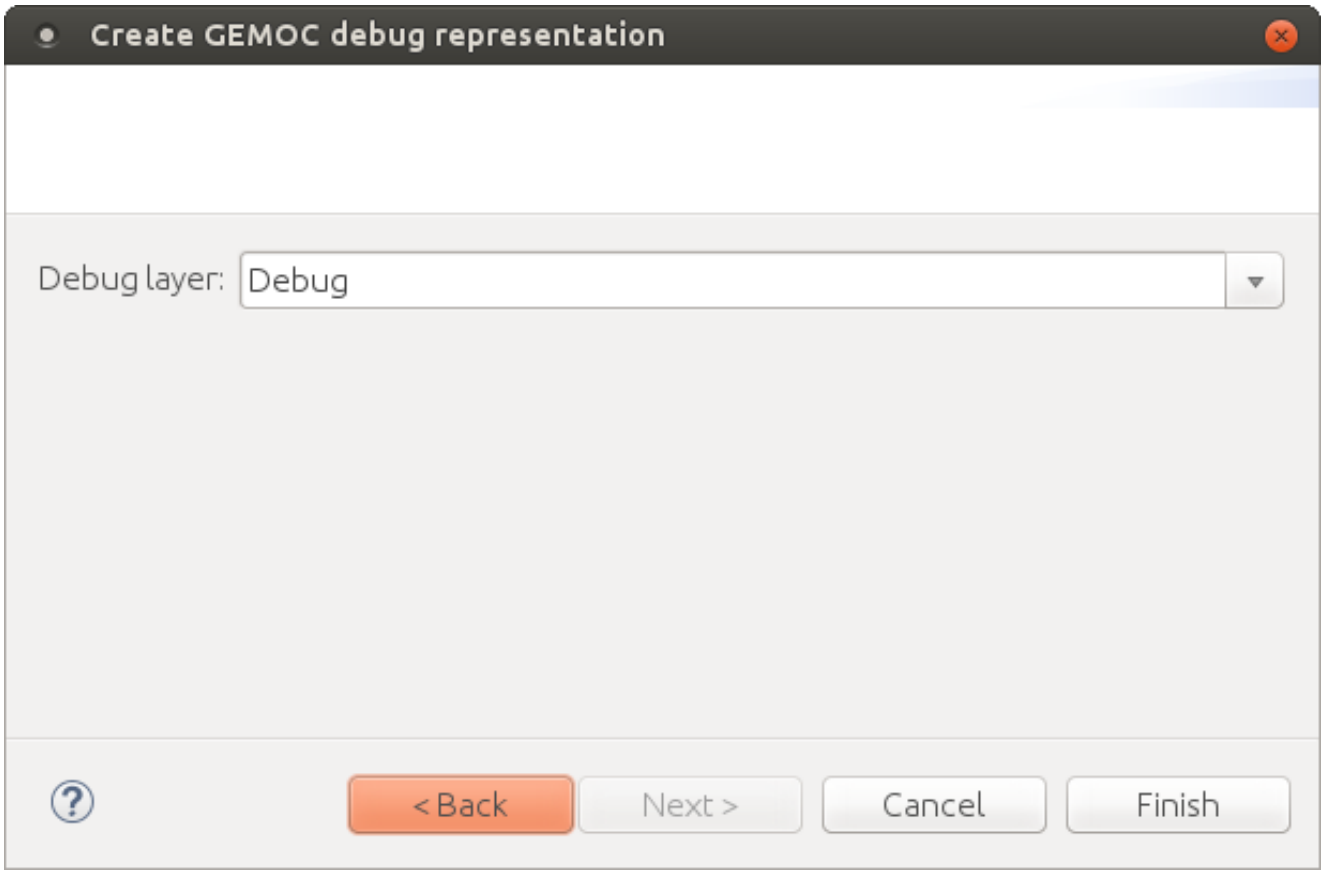

Create GEMOC debug representation
Debug layer: Debug
< Back
Next >
Cancel
Finish

Add a debug layer to an existing diagram description

It creates a default debug layer in an existing diagram representation. This should be used if you are also in charge of the language concrete syntax.

In this case, you can only select a diagram description from the workspace.





1.10.2. Implementation details

Implementation details are for advanced use and troubleshooting. It explains how the implementation works behind the scene. There are two main elements covered here : the debugger services class and the debug layer itself.

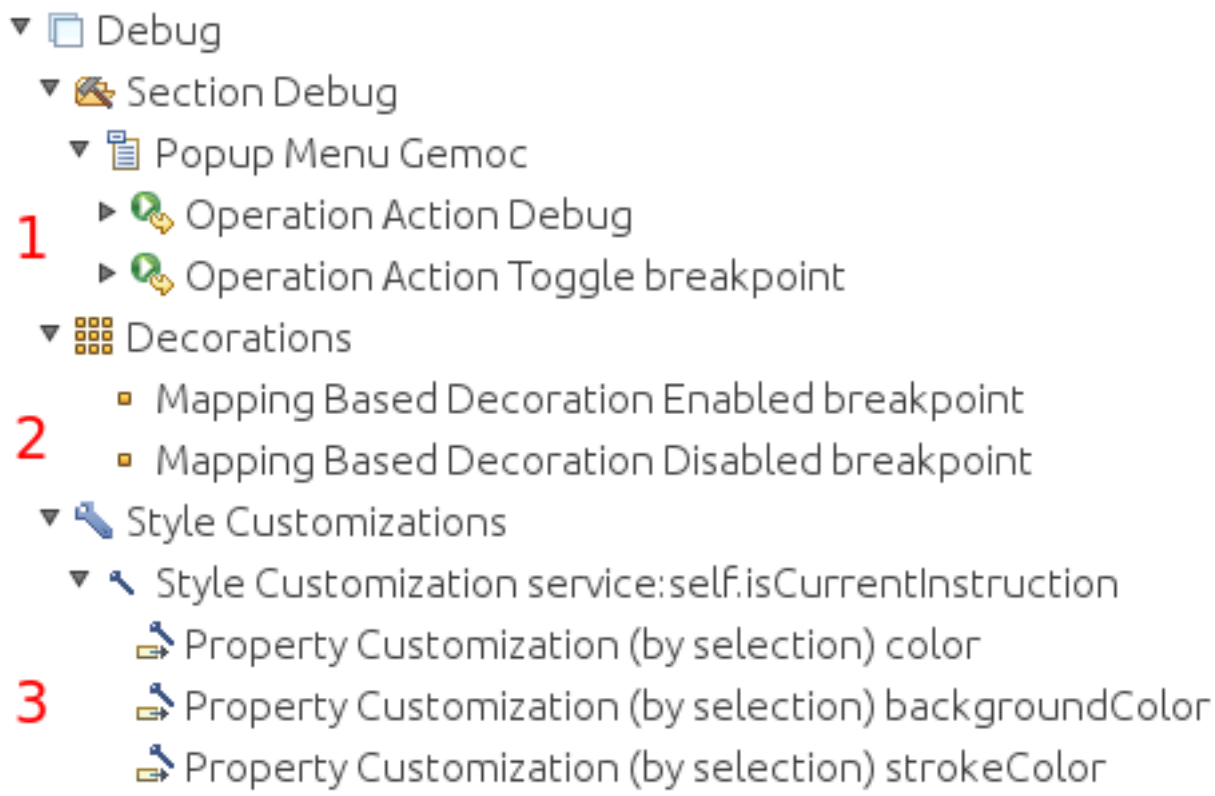
Debugger services

The debugger services class is use to tell which representations should be activated and refreshed during debug (see the getRepresentationRefreshList() method). It also provides a method to know if an element of the diagram is the current instruction. This is provided by the isCurrentInstruction() method.

```
public class TFSMDebuggerServices extends AbstractGemocDebuggerServices {

    @Override
    protected List<StringCouple> getRepresentationRefreshList() {
        final List<StringCouple> res = new ArrayList<StringCouple>();

        res.add(new StringCouple("TimedSystem","Debug"));
        res.add(new StringCouple("TFSM","Debug"));

        return res;
    }

}
```

Debug layer

The default debug layer adds action to start the simulation in debug mode and to toggle breakpoints (1). When a breakpoint exists for an element of the diagram, a visual feedback is displayed according to the breakpoint state (2). The current instruction is also highlighted in yellow by default (3).



This is a default debug layer, it can be customized to fit your needs. The customization use the Sirius description definition, see the Sirius Specifier Manual [http://www.eclipse.org/sirius/doc/specifier/Sirius%20Specifier%20Manual.html] for more details.

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.

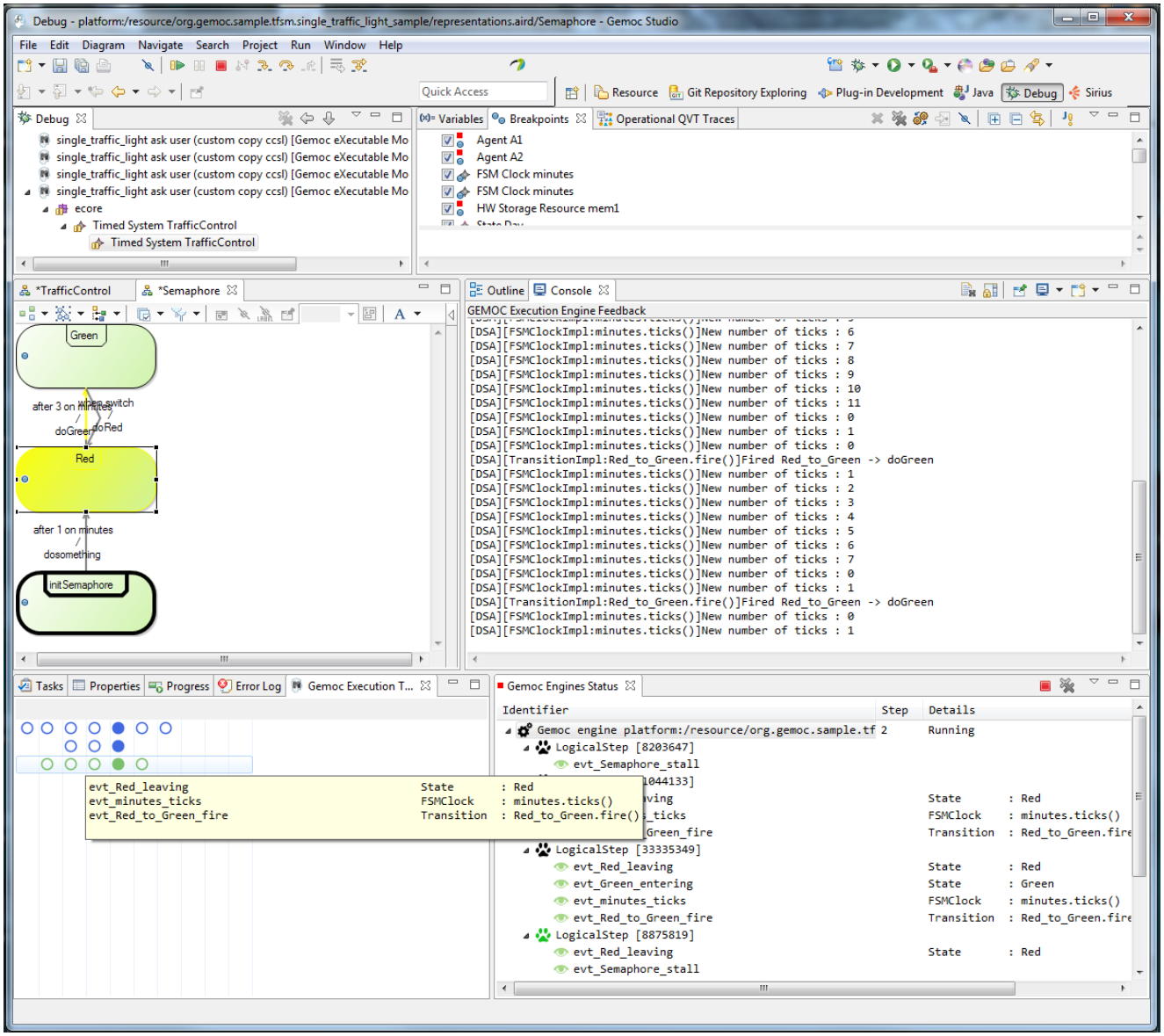


Figure 2.1. Screenshot of GEMOC Studio Modeling Workbench on the TFSM example (execution and animation).

2.2. Editing model

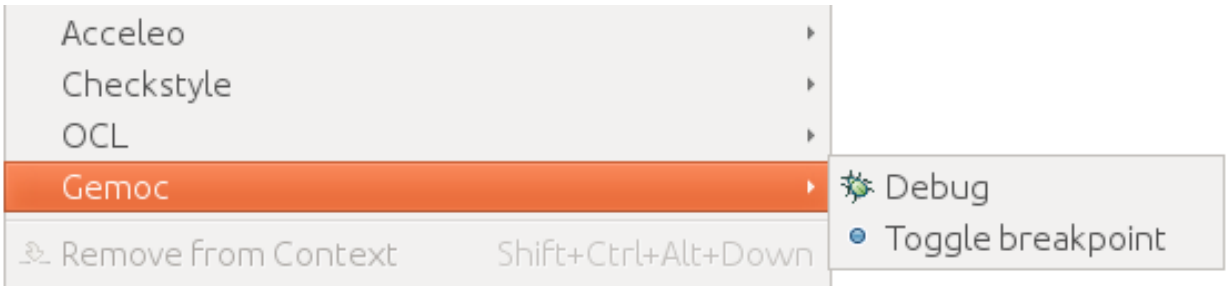
As seen in Section ???, an xDSML can provide different concrete syntaxes.

2.2.1. Editing model with Sirius

After defining your editor with Sirius (see Section 1.6.2, “Defining a Concrete Syntax with Sirius” for more details about Sirius), you can use your editor as described in the Sirius User Manual [http://www.eclipse.org/sirius/doc/user/Sirius%20User%20Manual.html].

Debug specific

If you have defined a debug representation using Section 1.10, “Defining a debug representation”. You can use the following actions to start a debug session and toggle breakpoints.



2.2.2. Editing model with EMF Tree Editor

TODO

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.

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.

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 with Automaton DSML

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 the same MoCC defined using MoCCML and a state machine
3. Second implementation with a focus on the MoCC (to show that a balance must be found)
4. Graphical visualization.
5. Nondeterministic automata (evolution of the previous example)
6. Pushdown Automata (demonstrate the feedback mechanism)
7. Integration of user defined code (connect the automata to a specific graphical interface).
8. 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

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.

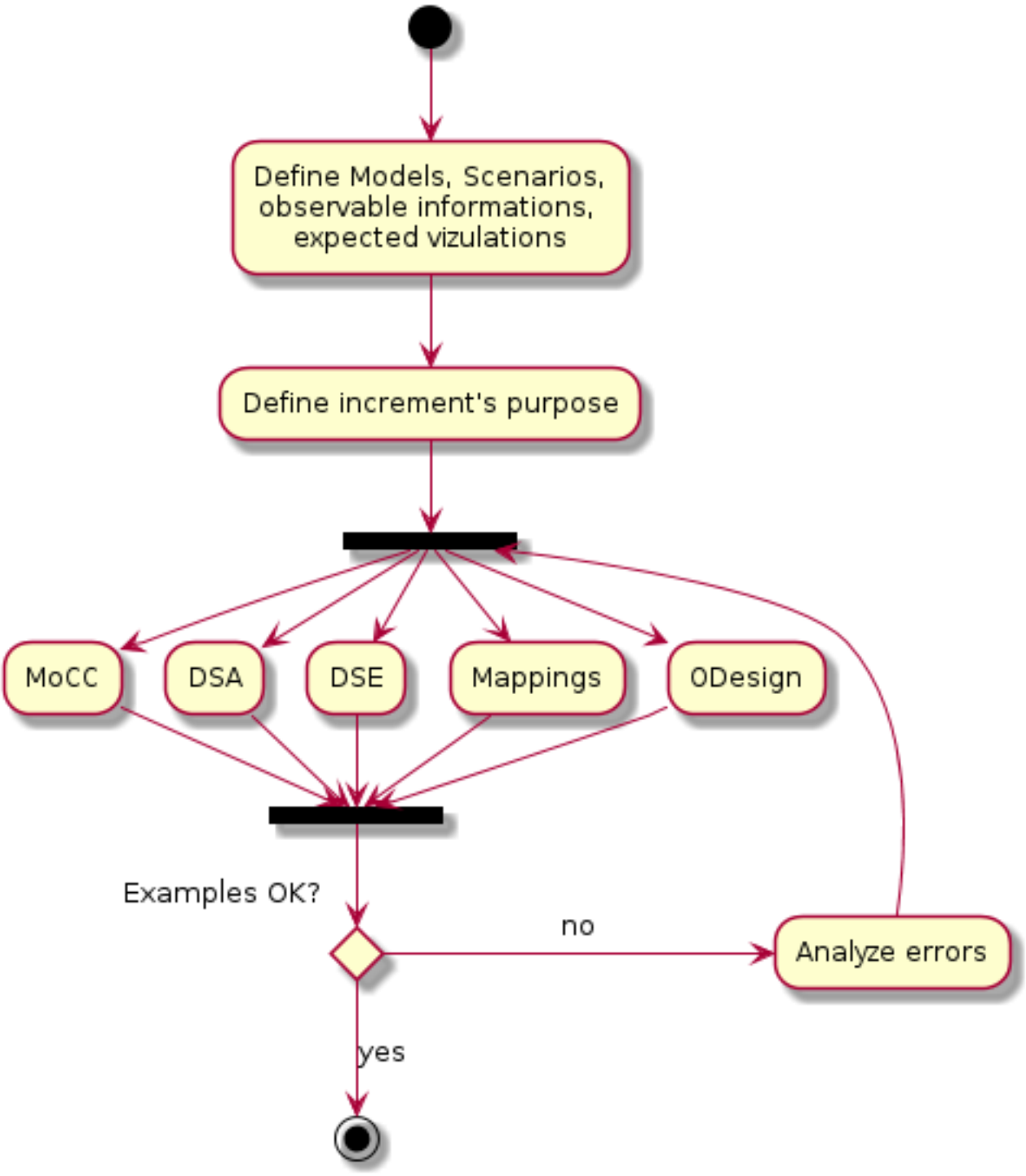


Figure 3.1. Overview of the GEMOC process

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).

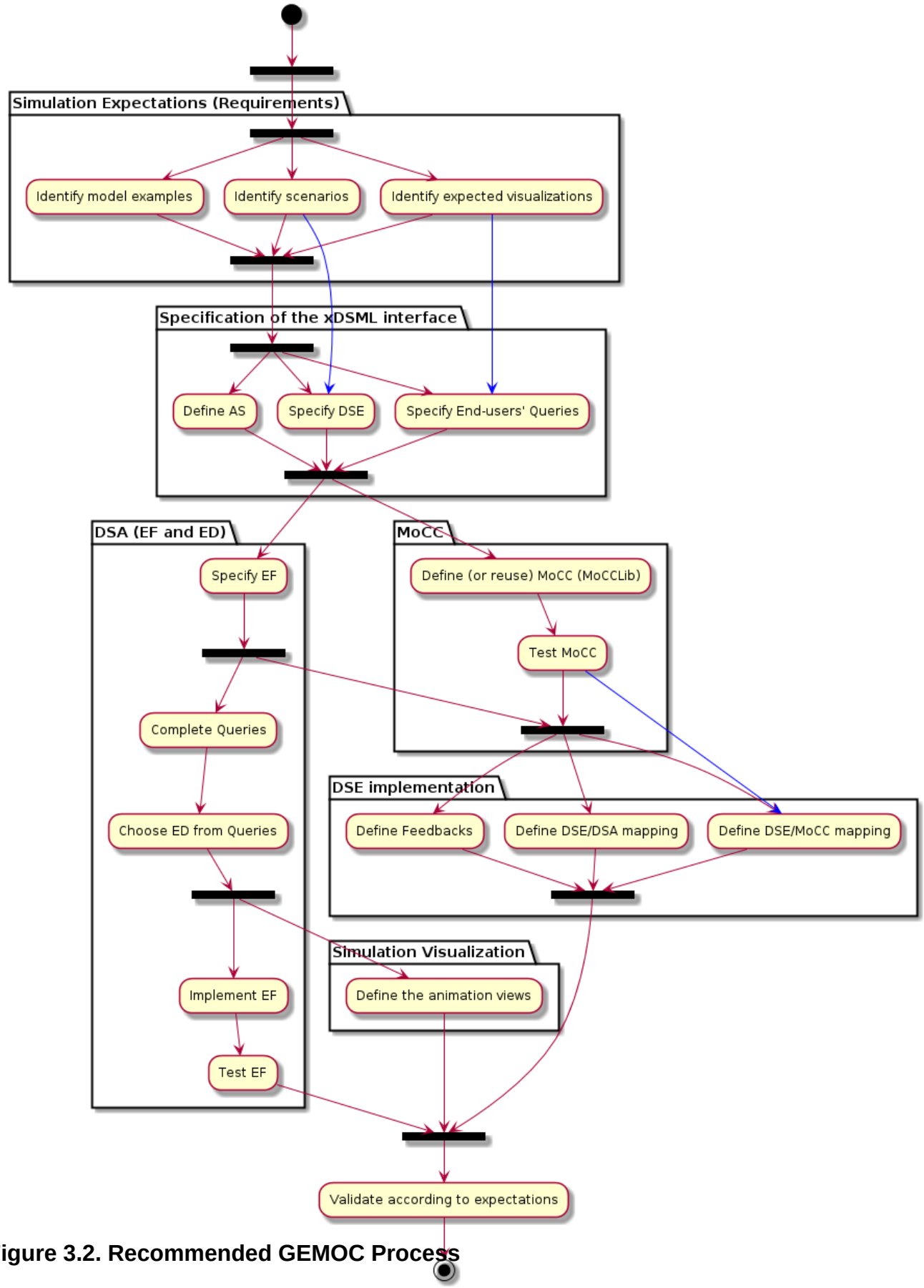


Figure 3.2. Recommended GEMOC Process

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.

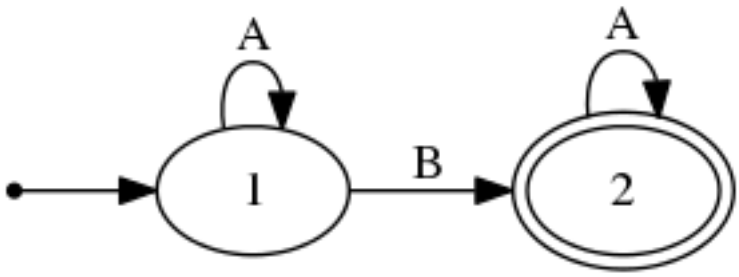


Figure 3.3. Automata which reads a*ba*

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.

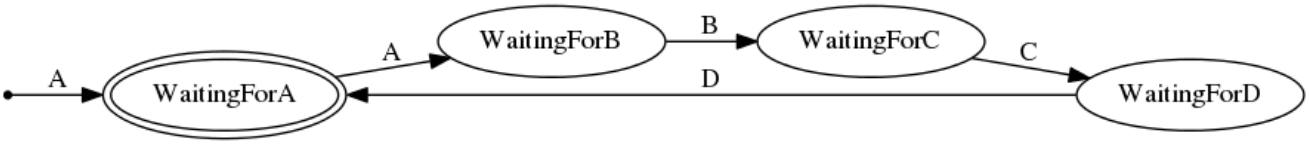


Figure 3.4. Automata which reads (ABCD)*

- 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:

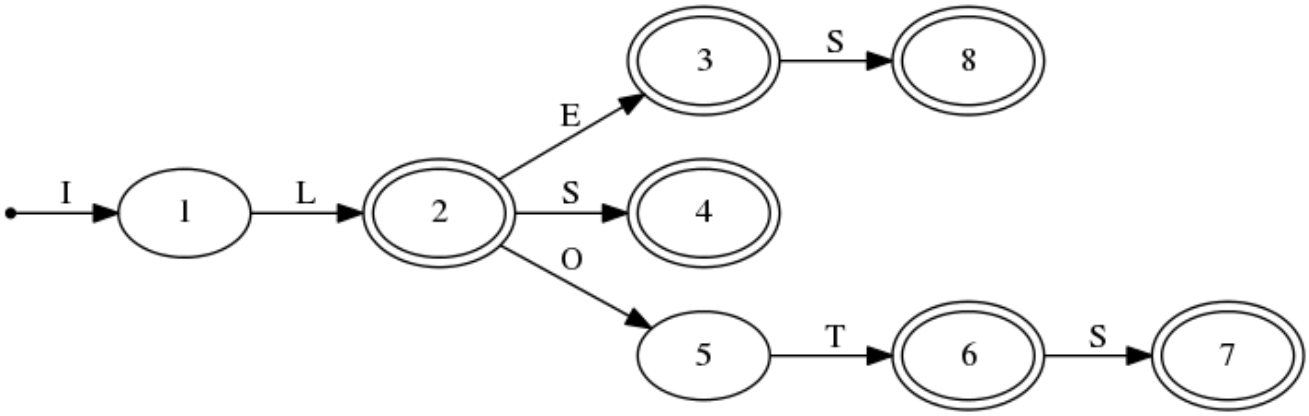


Figure 3.5. Automata which reads words

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?)

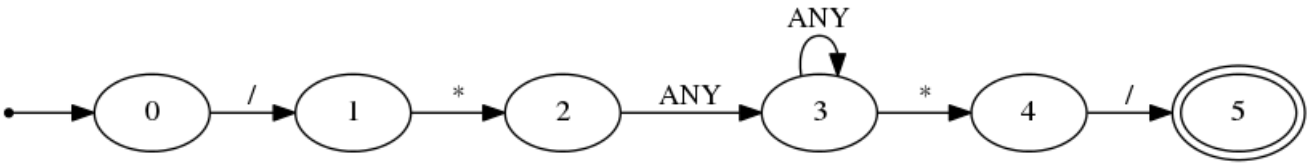


Figure 3.6. Automata which reads C commentary

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.

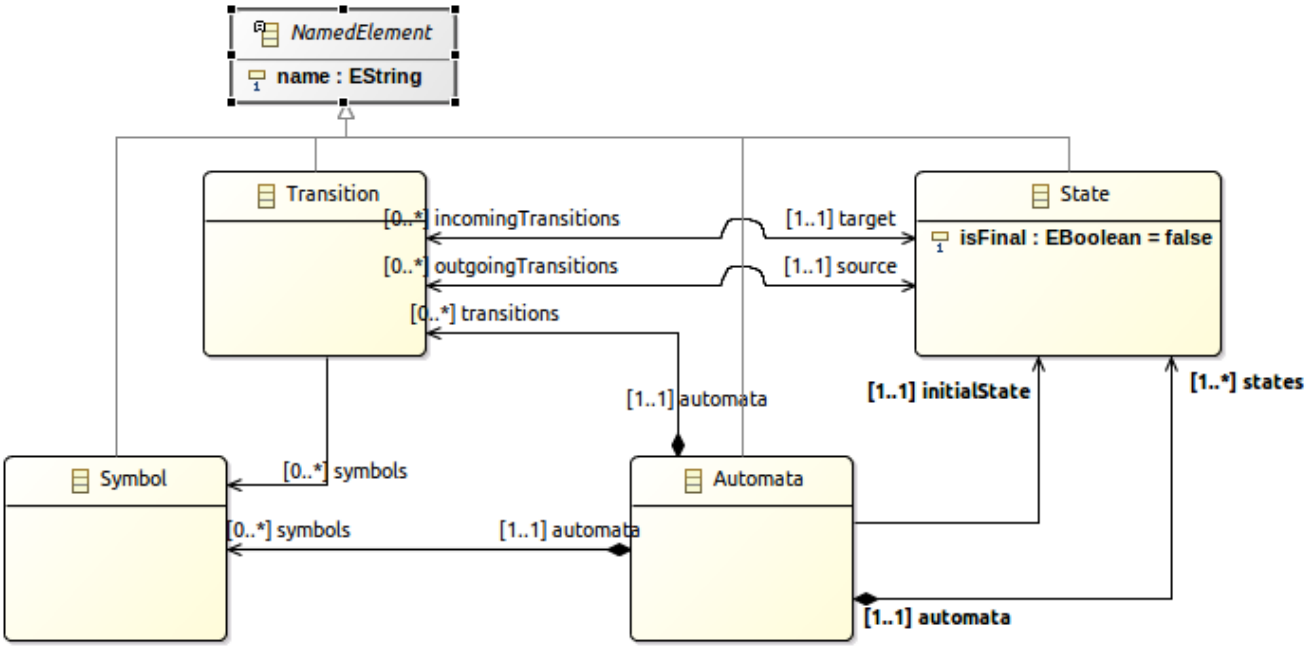


Figure 3.7. Automata Metamodel

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 *ECL project* in the DSE definition part 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.

```
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.

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: In the xDSML project, initialize the field "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 (using State Machines of MoCCML)

Same DSA

Same MoCC but expressed with a state machine and not CCSL relations.

3.7. Increment 3: 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.

3.8. Increment 4: Graphical visualization

- On the model (current states, the symbols already analyzed?)
- A tabular presentation ?
- A specific view as a Gantt

TODO: to be developed.

3.9. Increment 5: 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.

3.10. Increment 6: 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.

3.11. Increment 7: Call of user actions

Note

Will be added in a future version of this tutorial.

3.12. Increment: TBD

TODO: Which other increments?

3.13. 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

B

Both representations (graphical or textual) can be used for edition of models, 7

D

Decider, 24

E

ECL, 5
EMF Tree Editor, 24
Engine, 25, 25
Event Manager, 25, 25

F

Feedback Policy, 13
Feedback Specification, 12

G

GEL, 13
GEMOC
  process, 34
    overview, 30
GEMOC Dashboard, 2
GEMOC Language Workbench, 1

I

import, 5

L

Language Designer, 1, 2
Language Integrator, 1
Language Workbench, 1
Logical Step, 24, 27

M

MoCCML, 5
Modeling workbench, 23

O

overview, 30

P

process, 34
  overview, 30

R

Runtime Data, 25, 27

S

Sirius, 4, 14, 14, 14, 21, 24, 24
Sirius), 24

T

TFSM, 23
  Language Workbench, 1
  Modeling workbench, 23
Timeline, 25, 25