



Grant ANR-12-INSE-0011

# ANR INS GEMOC

## D2.2.1 - Model editor and Operational semantics of the MoCC modelling language (MoCCML )

**Task T2.2**

**Version 1.1**

# DOCUMENT CONTROL

| | –: 2014/09/28 | A: 2014/09/28 | B: | C: | D: |
|---|---|---|---|---|---|
| Written by<br><br>Signature | Joel Champeau | Papa Issa Diallo | | | |
| Approved by<br><br>Signature | | | | | |

| Revision index | Modifications |
|---|---|
| – | version 0.1 — Revised version |
| A | version 0.1 — Revised version with review comments |
| B | |
| C | |
| D | |

# Authors


| Author | Partner | Role |
| --- | --- | --- |
| Joel Champeau | Lab STICC / ENSTA Bretagne | Lead author |
| Julien DeAntoni | I3S / INRIA AOSTE | Contributor |
| Papa Issa Diallo | Lab STICC / ENSTA Bretagne | Contributor |
| Stephen Creff | Lab STICC / ENSTA Bretagne | Contributor |

# Contents



  

# 1. Introduction

In the D2.1.1 deliverable, we have presented the abstract syntax of the MoCC modeling language MoCCML . Several key concepts were introduced, thus giving an insight of the concepts that are used for the description of MoCC . The MoCC are used to provide the concurrency models of given DSLs. Indeed, with such concurrency models, the executable DSLs integrates formal behavioral semantics that control the execution of the DSL concepts. In order to use such concepts for MoCC instantiation, a MoCCML editor and solver were developed.

## 1.1 Purpose

This document presents the MoCCML editor implemented to support the edition of MoCC (in relation to the Task 2.1). In an effort to keep the defined models formal and to provide a solver for the MoCC , an operational semantics and a solver based on this formal semantics were also defined. These aspects are described in this document and an extended version of the formal operational semantics is provided in the D3.2.1 deliverable. Finally, a quick tutorial for using the MoCCML editor is described.

## 1.2 Perimeter

This document is the version v1.1 of the D2.2.1 dedicated to the description of the MoCCML editor, its operational semantics and MoCC solver. A more complete version of the operational semantics is provided in D3.2.1. The MoCC edition task is part of the language workbench activities. During this specific task, the MoCC are defined and further used to provide concurrency models to DSLs. Once the DSL models are defined and integrate MoCC properties, a MoCC dedicated solver is used to take into account the introduced concurrency semantics. Therefore, this deliverable will provide chapters that are important to understand the artifacts implemented (delivered $Software$) for MoCCML model edition and MoCCML model solver.

## 1.3 Definitions, Acronyms and Abbreviations

- **AS**: Abstract Syntax.
- **API**: Application Programming Interface.
- **Behavioral Semantics**: see *Execution semantics*.
- **CCSL**: Clock-Constraint Specification Language.
- **CS**: Concrete Syntax.
- **Domain Engineer**: user of the Modeling Workbench.
- **DSA**: Domain-Specific Action.
- **DSE**: Domain-Specific Event.
- **DSML**: Domain-Specific (Modeling) Language.
- **Dynamic Semantics**: see *Execution semantics*.
- **Eclipse Plugin**: an Eclipse plugin is a Java project with associated metadata that can be bundled and deployed as a contribution to an Eclipse-based IDE.
- **ED**: Execution Data (part of DSA).

- **EF**: Execution Function (part of DSA).
- **Execution Semantics**: Defines when and how elements of a language will produce a model behavior.
- **GEMOC Studio**: Eclipse-based studio integrating both a language workbench and the corresponding modeling workbenches.
- **GUI**: Graphical User Interface.
- **Language Workbench**: a language workbench offers the facilities for designing and implementing modeling languages.
- **Language Designer**: a language designer is the user of the language workbench.
- **MoCC**: Model of Concurrency and Communication.
- **Model**: model which contributes to the convent of a View.
- **Modeling Workbench**: a modeling workbench offers all the required facilities for editing and animating domain specific models according to a given modeling language.
- **MSA**: Model-Specific Action.
- **MSE**: Model-Specific Event.
- **RTD**: RunTime Data.
- **Static semantics**: Constraints on a model that cannot be expressed in the metamodel. For example, static semantics can be expressed as OCL invariants.
- **TESL**: Tagged Events Specification Language.
- **xDSML**: Executable Domain-Specific Modeling Language.

### 1.4 Summary

This deliverable presents two inputs related to the description of MoCC and their solver. The document will first provide an overview of MoCCML and how it fits in the GEMOC STUDIO process of making DSLs executable; then the document will present the implemented artifacts i.e. the MoCCML editor and the solver for the MoCC .

## 2. Explicit Concurrency modeling with MoCCML

The meta-language MoCCML tends to crystallize the best practices from the concurrency theory and the model-driven engineering. It leverages experiences on the explicit definition of the valid scheduling of an application through a clock constraint language [4] and an automata language [3]. It also reifies the appropriate concepts to enable automated reasoning.

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 specific constraints (named MoCC on Figure 2.1 and conforming to MoCCML ). These constraints can also be of a different kind, for instance to express a deadline, a minimal throughput or a hardware deployment. They are eventually instantiated to define the execution model of a specific model (see Figure 2.1). The execution model is a symbolic representation of all the acceptable schedules for a particular model.

To enable the automatic generation of the execution model, the MoCC is weaved into the context of specific concepts from the abstract syntax of a DSL. This contextualization is defined by a mapping between the elements of the abstract syntax and the constraints of the MoCC (achieved by the box named *Mapping* in Figure 2.1). The mapping defined in

 

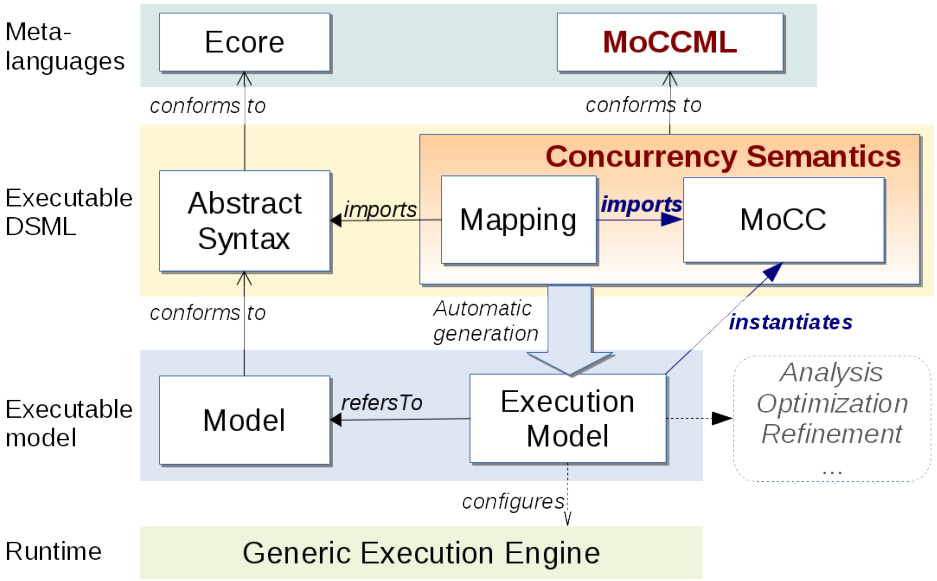


*Figure 2.1:* MoCCML *Big Picture*

MoCCML is based on the notion of event, inspired by ECL [2], an extension of the Object Constraint Language [5]. The separation of the mapping from the MoCC makes the MoCC independent of the DSL so that it can be reused.

From such description, for any instance of the abstract syntax it is possible to automatically generate a dedicated execution model (see "executable model" in Figure 2.1).

In our approach, this execution model is acting as the configuration of a generic execution engine (see "generic execution engine" in Figure 2.1), which can be used for simulation or analysis of any model conforming to the abstract syntax of the DSL.

MoCCML is defined by a metamodel (*i.e.,* the abstract syntax) associated to a formal Structural Operational Semantics [6]. MoCCML also comes with a model editor combining textual and graphical notations, as well as analysis tools based on the formal semantics for simulation or exhaustive exploration.

# 3. MoCCML Editor and Executable Model Solver

## 3.1 MoCCML Syntax

### *3.1.1 Abstract Syntax*

MoCCML is based on the principle of defining constraints on events. In the abstract syntax, there are two categories of constraint definitions: the *Declarative Definition*s and the *Constraint Automata Definition*s (see Figure 3.1). Each constraint definition has an associated *ConstraintDeclaration* that define the prototype of the constraint. These definitions constraint some *Event*s.

A declarative definition is defined as a set of constraint instances. For more details, we refer the reader to [1] that described the declarative part inspired from the CCSL language.

As illustrated in Figure 3.1, a *Constraint Automata Definition* contains a set of *States* with a single initial state and one or more final states. It also contains *DeclarationBlocks* where local *Variable*s can be declared. To ease exhaustive simulations see D3.4.1, we restricted the types of the variables (and parameters to be Event or Integer).

The constraint automata definition introduces the concept of Transition which links a *source* state and a *target* state. It contains a *Trigger* that defines two sets of events (namely *trueTriggers* and *falseTriggers*). The transition is fired if the events in the *trueTriggers* set are present and the ones in the *falseTriggers* set are absent. A transition can define a *Guard*. A guard is a boolean expression over the local variables or the parameters of the definition. Finally, during the

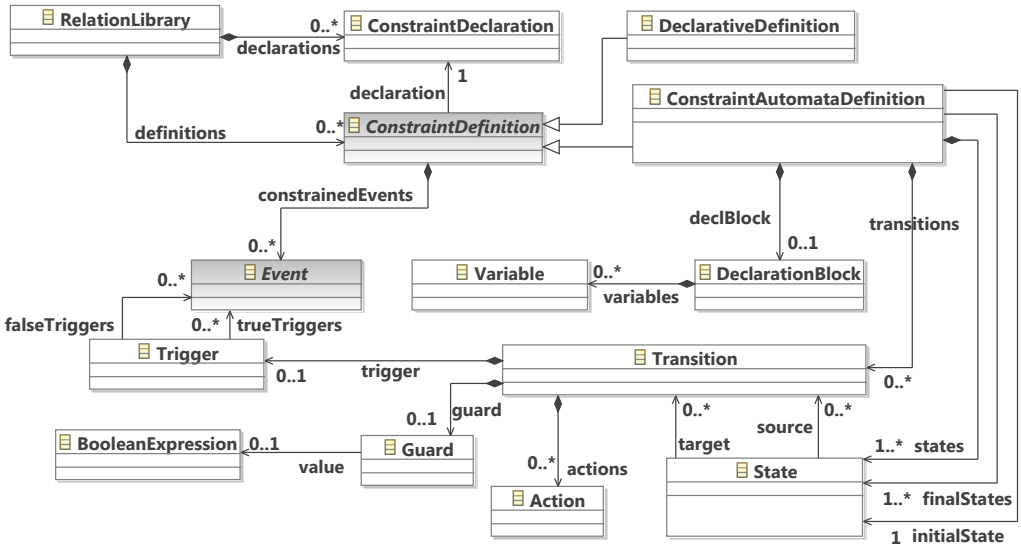


*Figure 3.1: Excerpt of* MoCCML *Metamodel*

firing of a transition, actions such as integer assignments (possibly with a value resulting from an expression such as the increment of a counter) can operate on the local variables.

### *3.1.2 Concrete Syntax and 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 conforming to the aforementioned abstract syntax.

The graphical model shown in Figure 3.2 defines a sample of the kind of editor that we want to put in place for the MoCCML concrete syntax editor. The example shows a MoCC Constraint Library (*SimpleSDFRelationLibrary*), which contains a constraint declaration named *PlaceConstraint*. The constraint declaration is associated to a constraint automata definition (*PlaceConstraintDef*). In this library, we define a constraint between the *read* and *write* events. The automaton operates on 5 integer parameters (one variable: *size*; and 4 constants: $itsCapacity$, $itsDelay$, $pushRate$, $popRate$), which are set during the instantiation process.

The graphical syntax is realized using the *Sirius* framework and the textual syntax using *Xtext*. Both graphical and textual syntaxes are defined using the code generated from the EMF GENMODEL of the MoCCML abstract syntax. In the following sections, we describe the implemented artifacts.

#### Graphical Concrete Syntax and Editor

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 relations); another for the implementation of the relations in the form of automata.

For instance, the first level of representation contains elements as illustrated in Figure 3.3. 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. The graphical model defines a new MoCC Relation Library (*MoCA*). In *MoCA*, we define one MoCC implementation, declared as a new Relation Declaration (*FirstFSM_Decl*). The declaration contain several formal parameters (*param1*, *clk1*, *clk2*) specified with (Declaration Parameter ).

Each defined Relation Declaration is associated to a State-Based Definition (*FirstFSM_Impl*). The association is provided by Set Declaration Relation . This representation is similar to the expected format from Figure 3.2. The State-Based Definition (*FirstFSM_Impl*) is the connection point to the second level of graphical representation to define

 

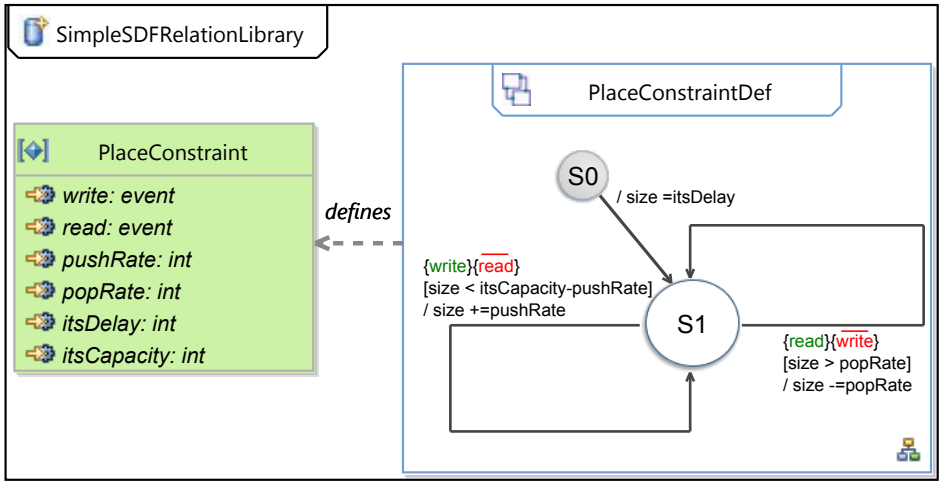


*Figure 3.2: Overview of the* MoCC *Library Graphical Representation*

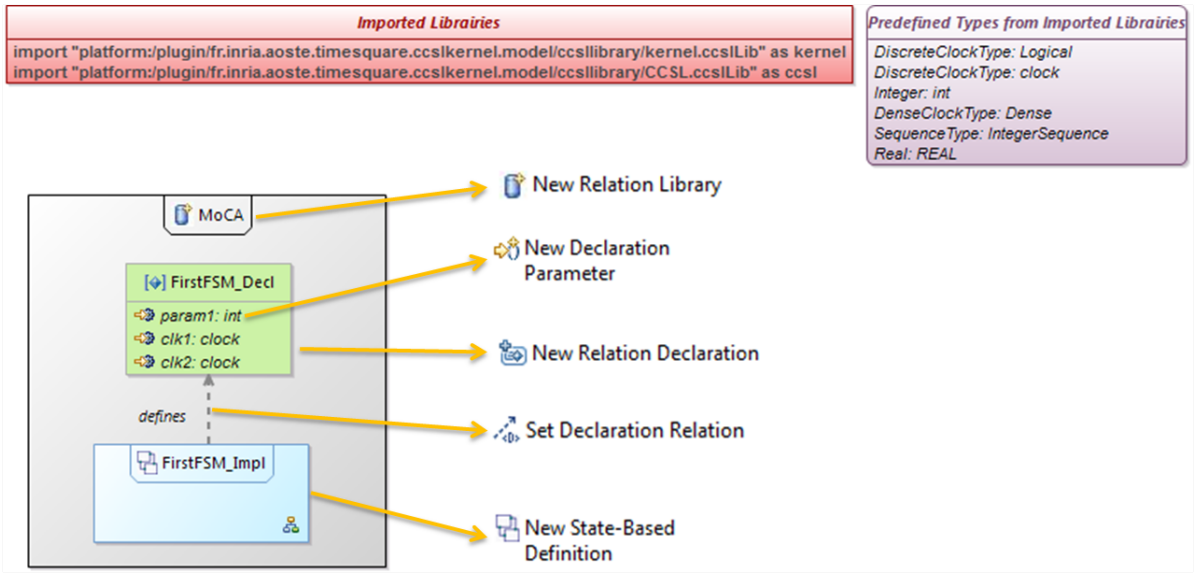


*Figure 3.3: First Level of graphical* MoCCML *model Representation*

*Table 3.1: Description of the Top level* MoCC *modeling key elements*

| Relation Library Graphical Syntax | Description |
|---|---|
| Relation Library | Syntax used to declare a new MoCC *RelationLibrary* |
| Relation Declaration | Syntax used to declare the interface of a *StateBasedRelationDefinition* ie a new *RelationDeclaration* |
| Declaration Parameter | Syntax used to declare formal parameters for a *RelationDeclaration* eg *AbstractAction* |
| State-Based Definition | Syntax used to declare a *StateBasedRelationDefinition* |
| Set Declaration Relation | Syntax used to define the association of a *StateBasedRelationDefinition* to its *RelationDeclaration* |

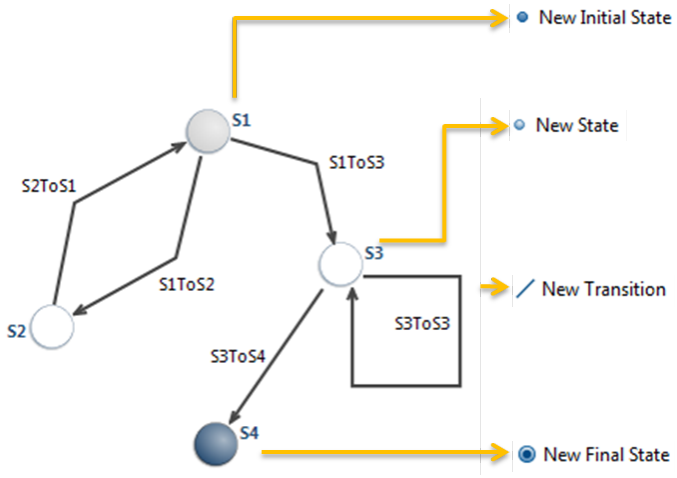


*Figure 3.4: Second Level of graphical* MoCCML *model Representation*

automata. Before describing the second level of representation, we propose Table 3.1 to summarize the graphical syntax key elements of the first level of representation.

The second level of graphical representation is linked to the syntax element State-Based Definition that instantiates *StateBasedRelationDefinition* [1]. As shown in Figure 3.4, a *StateBasedRelationDefinition* is specified as an automaton. An automaton defines a Initial State , several new State and Final State and Transition to link the states. The actions and guards on the transitions are provided in the form of textual concrete syntax that will be described in section 3.1.2. Table 3.2 summarizes the graphical syntax elements defined at the second graphical representation level.

The textual syntax is integrated with the graphical syntax at different connection points. For the two level of representation, we have defined the connection points for textual syntax as shown in Figures 3.5 and 3.6. The textual editor is launched by double-click on the graphical representation of the concept.

[1] *StateBasedRelationDefinition* refers to *Constraint Automata Definition*

*Table 3.2: Description of the graphical key elements for* StateBasedRelationDefinition

| *StateBasedRelationDefinition* Graphical Syntax | Description |
|---|---|
| Initial State | Syntax used to declare a new MoCCML initial *State* |
| State | Syntax used to declare new MoCCML *State*s |
| Final State | Syntax used to declare new MoCCML final *State*s |
| Transition | Syntax used to declare new MoCCML *Transition*s |

  

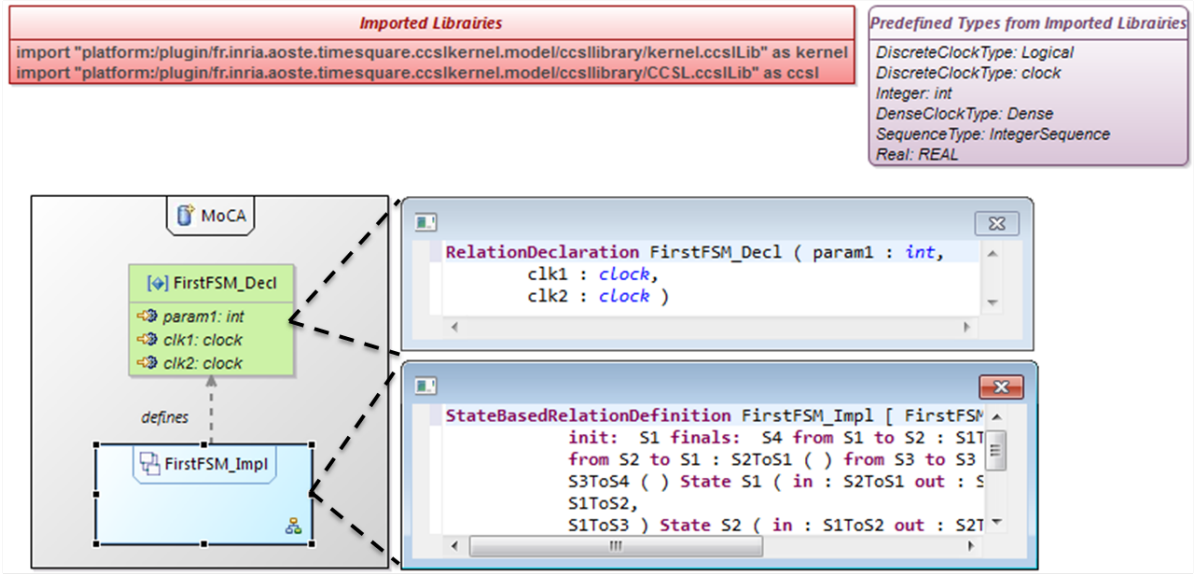


*Figure 3.5: Connection point of textual syntax in graphical Representation (Level 1)*

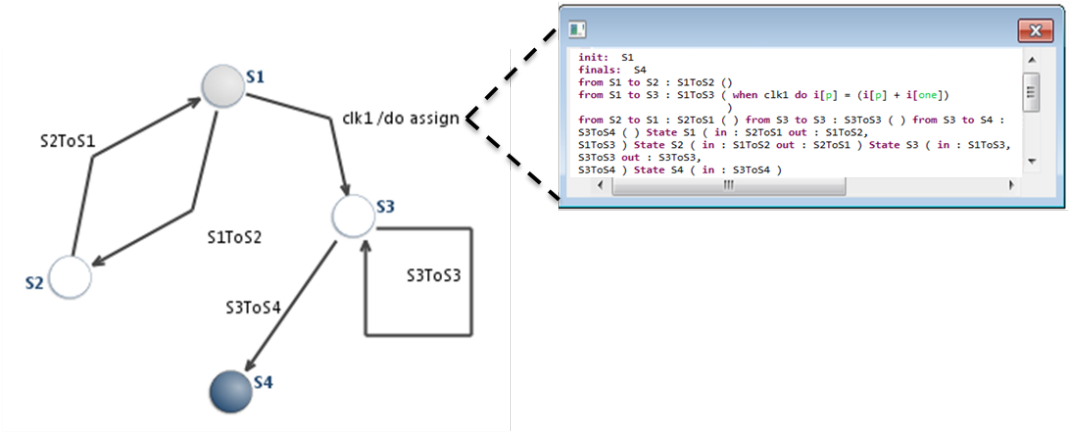


*Figure 3.6: Connection point of textual syntax in graphical Representation (Level 2)*

In the first level of representation, the connection points are *RelationDeclaration* [2] and *StateBasedRelationDefinition* . In the second level of representation, the connection point is on the *Transition* elements.

The mixed graphical/textual model can be serialized as a single textual model. The next Section, will give details on the textual syntax of the language.

Textual Concrete Syntax and Editor

The description of the textual concrete syntax is based on the previous *MoCA* example. We will consider the whole syntax of the model which is shown by Listing 3.1.

In this Listing, we can see that keywords similar to those described in the graphical part are used to define the library of MoCC . We have keywords such as *StateRelationBasedLibrary* (MyLib), *RelationLibrary* (*MoCA*), *StateBasedRelationDefinition* .

Using the textual syntax, several aspects related to variable declaration can be written. For instance:

[2] refers to *ConstraintDeclaration*

– At line 22, the local declaration and initialization of parameters (Integer p = 0, Integer z = 0). The declaration of named guards comparing p to z (*pSupZero i[p] > i[z]*).

– At line 31, the syntax defines two states and their transition relation (from S1 to S3: S1ToS3). The transition has a trigger, a guard and an action. Each declared Transition starts with a keyword *from*, followed by the two related states and the name of the transition as shown at line 22.

```
1  StateRelationBasedLibrary MyLib
2  {
3  imports
4    {
5      import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/kernel.ccslLib" as
           kernel;
6      import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/CCSL.ccslLib" as
           ccsl;
7    }
8
9    RelationLibrary MoCA
10   {
11     RelationDeclaration FirstFSM_Decl (
12     param1 : int ,
13     clk1 : clock,
14     clk2 : clock,
15     clk3 :Dense start )
16
17     StateBasedRelationDefinition FirstFSM_Impl [ FirstFSM_Decl ]
18     {
19
20       Declarations
21       {
22        Integer  p = 0
23        Integer  z = 0
24          (#ref pSupZero i[p] > i[z])
25       }
26
27       init:  S1
28       finals:  S4
29
30       from S1 to S2 : S1ToS2 ()
31       from S1 to S3 : S1ToS3 ( when clk1 if [pSupZero] do i[p] = (i[p] + i[one]) )
32       from S2 to S1 : S2ToS1 ( )
33       from S3 to S3 : S3ToS3 ( )
34       from S3 to S4 : S3ToS4 ( )
35
36       State S1 ( in : S2ToS1 out : S1ToS2,S1ToS3 )
37       State S2 ( in : S1ToS2 out : S2ToS1 )
38       State S3 ( in : S1ToS3, S3ToS3 out : S3ToS3, S3ToS4 )
39       State S4 ( in : S3ToS4 )
40     }
41   }
42 }
```

*Listing 3.1: Excerpt of edited Textual* MoCCML *Model*

Within a defined Transition, a trigger is preceded by the keyword *when* trigger (here *when clk1*); the guard is preceded by the condition primitive *if* (here *if [pSupZero]* ie $p > z$); and the action is preceded by the keyword *do* (here *do i[p] = (i[p] + i[one] i.e.,* $p = p+1$). All the relations between the state and transitions are described following such syntactical rules implemented in the grammar. The syntax may be improved in updated versions of the Editor.

Table 3.3 summarizes the key elements of the textual syntax and their descriptions.

  

*Table 3.3: Textual Syntax key elements to define a* StateRelationBasedLibrary

| Textual Syntax | Description |
|---|---|
| **StateRelationBasedLibrary** MyLib | Syntax used to declare new automata MoCCML libraries; (keyword + library name) |
| **import** "lib-path" **as** libname; | Syntax used to import existing MoCC libraries (currently CCSL or MoCCML libraries) |
| **RelationLibrary** MoCA | Syntax used to declare a new automata MoCCML library; (keyword + library name) |
| **RelationDeclaration** FirstFSM_Decl ( param1 : *int*, clk1 : *clock*) | Syntax used to declare the interfaces of a *StateBasedRelationDefinition* and their formal parameters |
| **StateBasedRelationDefinition** FirstFSM_Impl $[FirstFSM\_Decl]$ | Syntax used to declare a new automata MoCCML relation definition; (keyword + library name + interface name) |
| **Declarations** *Integer* p = 0 (**#ref** pSupZero i[p] $>$ i[z]) | Syntax used to declare local parameters in a *StateBasedRelationDefinition* |
| **init**: S1 **finals**: S4 | Syntax used to declare an initial state and final states |
| **from** S1 **to** S3 : S1ToS3 ( **when** clk1 **if** [pSupZero] **do** i[p] = (i[p] + i[one])) | Syntax used to declare relation between states (transition), trigger, guard and action for the transition |
| **State** S1 (**in**: S2ToS1 **out**: S1ToS2, S1ToS3) | Syntax to declare for a given state (S1), its input and output transitions |

### 3.2 MoCCML operational semantics and implemented Solver

The MoCCML semantics provides the way in which each MoCCML element should be executed. This semantics is also useful to implement a solver for the MoCC to be integrated with an DSL abstract syntax (see Figure 2.1).

This section overviews the operational semantics of MoCCML , which allows the effective construction of the acceptable schedules. The interested reader can refer to [1] or D3.2.1 for a full definition of the operational semantics. An execution model consists in a finite set of discrete events, constrained by a set of constraints. A *schedule* $\sigma$ over a set of events $E$ is a possibly infinite sequence of *Steps*, where a step is a set of occurring events. $\sigma : \mathbb{N} \rightarrow 2^E$. For each step, one or several event(s) can occur. The goal of the semantics rules is to specify how to construct the acceptable schedules.

The semantics of a specification expressed in MoCCML is given as a Boolean expression on $\mathscr{E}$, where $\mathscr{E}$ is a set of Boolean variables in bijection with $E$. For any $e \in \mathscr{E}$, if $e$ is valued to *true* then the corresponding event occurs; if valued to *false* then it does not occur. If no constraints are defined, each boolean variable can be either true or false and there are $2^n$ possible futures for all steps, where $n$ is the number of events. Consequently, in this case the number of acceptable schedules is infinite.

Each time a constraint is added to the specification, it adds boolean constraints on $\mathscr{E}$. The boolean constraints depends on the definition of the MoCCML constraint and its internal state. When several MoCCML constraints are defined, their boolean expressions are put in conjunction so that each added constraint reduces the set of acceptable schedules. For instance, if the *sub-event* declarative constraint is defined between two events e1 and e2 (*i.e.,* e1 subevent of e2), then the corresponding boolean expression is $\mathrm{e1} \Rightarrow \mathrm{e2}$.

The same principle applies to the constraint automata definitions. The boolean expression associated to a specific constraint automata is obtained according to: 1) the value of the automata local variables; 2) the current state; 3) the evaluation of boolean guards on the output transition of the current state and 4) the triggers ($trueTriggers$ and $falseTriggers$) on the output transitions of the current state.

The semantics of a constraint automata is defined as a *logical disjunction* of the boolean expressions associated to the output transitions of the current state. For a transition $t$, if its guard is valued to true, the resulting boolean expression is the conjunction of all the events in the $trueTrigger$ set in conjunction with the conjunction of the negation of all the events in the $falseTrigger$ set. For instance, in the constraint automata depicted in Figure 3.2, the boolean expression when $size$ is lesser than $itsCapacity$ minus $pushRate$ is: $write \wedge \neg read$. In the case where $size$ is also greater than $popRate$ the automata semantics is $(write \wedge \neg read) \vee (read \wedge \neg write)$. If the new computed step is such that the boolean equation of one transition is valued to true, then the transition is fired, meaning that the current state evolves to the target of the fired transition and the actions of this transition are executed.

A MoCCML model solver was implemented based in the above described semantics. The solver is defined as an extension of the TimeSquare solver for CCSL. The extension takes into account contraints described as MoCCML automata definitions and add new branches in the Binary Decision Graph that is built by the TimeSquare solver thus representing the new constraints induced by the MoCCML automata.

### 3.3 Deployed Plugins in the GEMOC STUDIO

This section gives a list of the implemented plugin projects corresponding to the above described contributions. The projects can be found in (*org/gemoc/MoCC/*) of the GEMOC GIT collaborative development platform.

The abstract syntax of MoCCML is divided into two connected parts i.e. *ccslmocc* and *fsmkernel*[3]. As a remainder, the *ccslmocc* contains the concepts used to declare MoCC libraries (eg, *StateRelationBasedLibrary* ); and the *fsmkernel* contains the concepts for the description of the MoCCML automata stored in the MoCC libraries. This part of MoCCML is connected to the *ccslmocc* through relations between concepts such as *AbstractAction* and *FinishClock* .

The plugin implementing the abstract syntax are provided in *org/gemoc/MoCC/AS*:

- org.gemoc.mocc.ccslmocc.model
- org.gemoc.mocc.ccslmocc.model.edit
- org.gemoc.mocc.ccslmocc.model.editor

[3]For more details on the relations between *ccslmocc* and *fsmkernel*, the reader can refer to the D2.1.1.

  

- org.gemoc.mocc.fsmkernel.model
- org.gemoc.mocc.fsmkernel.model.edit
- org.gemoc.mocc.fsmkernel.model.editor

These plugin projects are used to create the graphical and textual editors respectively in *Sirius* and *Xtext*. The source plugin projects for the description of the graphical and textual editors are located in the following repositories:

- *org/gemoc/MoCC/GraphicalCS*
  - org.gemoc.mocc.ccslmocc.model.design
  - org.gemoc.mocc.fsmkernel.model.design
- *org/gemoc/MoCC/TextualCS*
  - org.gemoc.mocc.ccslmocc.model.xtext.mocdsl
  - org.gemoc.mocc.ccslmocc.model.xtext.mocdsl.sdk
  - org.gemoc.mocc.ccslmocc.model.xtext.mocdsl.tests
  - org.gemoc.mocc.ccslmocc.model.xtext.mocdsl.ui
  - org.gemoc.mocc.ccslmocc.model.xtext.fsmdsl
  - org.gemoc.mocc.ccslmocc.model.xtext.fsmdsl.sdk
  - org.gemoc.mocc.ccslmocc.model.xtext.fsmdsl.tests
  - org.gemoc.mocc.ccslmocc.model.xtext.fsmdsl.ui

The two *Sirius* projects in *org/gemoc/MoCC/GraphicalCS* contain the viewpoint specification files for: the graphical representation of the MoCC libraries and the graphical representation of each *StateRelationBasedLibrary* . The *Xtext* projects in *org/gemoc/MoCC/TextualCS* contain, mainly, the grammar of the textual concrete syntax including keywords of the syntax. The extension of the MoCCML model files is "*.moccml*".

The implementation of the solver extension for MoCCML models is located in the repository *org/gemoc/MoCC/solver/* and the implemented plugin project is identified as follows:

- fr.inria.aoste.timesquare.ccslkernel.solver.extension.statemachine

## 4. Quick start tutorial for MoCCML Model Edition

This chapter gives basic guidelines to create MoCCML models in the GEMOC STUDIO . The MoCC are created using the file extension "*.moccml*". A starting point could be to use the xDSML project to create a new MoCC project: right click on the xDSML project, go to the menu item "GEMOC Language" and select *Create MoC Project* as shown in Figure 4.1.

A wizard will help to create a first MoCC project with a default MoCC editor file. This file can be edited directly in a textual style using keywords such as described in section 3.1.2 to define new MoCC libraries, new declarations and definitions in a given library.

The file can also be edited in a graphical style using the graphical editor. To do so, right-click on the MoCC file and select New → Representations File. From the new generated set of file for graphical edition, select a root element in the MoCC file to create a new graphical view (new Diagram). A Toolset proposing the editor elements ( see Figure 3.3 and 3.4) are proposed to create the libraries and so on.

A predefined example of a project implementing a MoCCML model is available in the GIT at

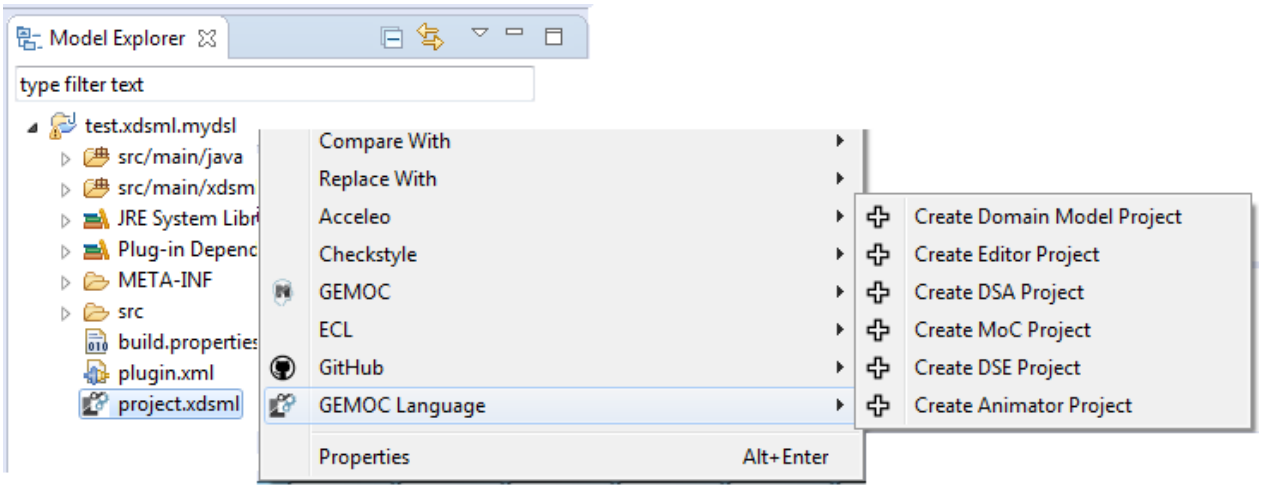


*Figure 4.1: Screenshot of a* MoCC *Project creation process*

*org/gemoc/sample/SigPML_MoCCML/* with a ReadME file describing the steps to use the MoCC files.

## 5. Conclusion and Perspectives

This version of the editor is based on the MoCCML metamodel presented in D2.1.1. The MoCC editor provides the opportunity to create MoCC definitions in a graphical and textual way. The models can be tested after integration with a DSL and using the implemented extended solver for TimeSquare. By test, we refer to the execution of the models integrating MoCC constraints. Indeed, TimeSquare provides support for simulation and execution trace extraction from the set of possible execution paths. The set of all possible execution paths can be exhaustively provided using exhaustive exploration tools (see D 3.4.1). In fact, it could be interesting to have the exhaustive set of all possible execution paths for activities such as model-checking that verifies system properties. The current $Software$ version can be updated depending on feedbacks from the users of the editor.

## 6. References

[1] Julien Deantoni, Papa Issa Diallo, Joël Champeau, Benoit Combemale, and Ciprian Teodorov. Operational Semantics of the Model of Concurrency and Communication Language. Research Report RR-8584, September 2014.

[2] Julien Deantoni and Frédéric Mallet. ECL: the Event Constraint Language, an Extension of OCL with Events. Technical Report RR-8031, INRIA, 2012.

[3] Papa Issa Diallo, Joël Champeau, and Vincent Leilde. Model based engineering for the support of models of computation: The cometa approach. In *MPM*, 2011.

[4] Frédéric Mallet, Julien DeAntoni, Charles André, and Robert de Simone. The clock constraint specification language for building timed causality models - application to synchronous data flow graphs. *ISSE*, 6(1-2):99–106, 2010.

[5] OMG. *Object Constraint Language*, 2014. Version 2.4.

[6] Gordon D Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 1981.