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 2.0

DOCUMENT CONTROL

	–: 2014/09/28	A: 2014/09/28	B: 2015/04/30	C:	D:
Written by Signature	Joel Champeau	Papa Issa Diallo	Papa Issa Diallo		
Approved by Signature					

Revision index	Modifications
–	version 0.1 — Revised version with review comments
A	version 1.1 — Description of the implemented Software Graphical and Textual Editor and solver
B	version 2.1 — Description of the last modifications for the Graphical and Textual Editor
C	
D	

Authors

Author	Partner	Role
Joel Champeau	Lab STICC / ENSTA Bretagne	Lead author
Papa Issa Diallo	Lab STICC / ENSTA Bretagne	Contributor
Julien DeAntoni	I3S / INRIA AOSTE	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 how the metamodel was defined. The MoCCs are used to provide the underlying concurrency models of DSLs. Indeed, with such concurrency models, the executable DSLs integrates formal behavioral semantics that control the execution of the DSL concepts. In order to instantiate MoCCs, a MoCCML editor was developed. This document presents the editor as well as a solver for the MoCCs.

1.1 Purpose

This document presents the MoCCML editor implemented to support the edition of MoCCs (in relation to the Task 2.1) and its associated solver (which is an extension of the TimeSquare solver for automata). To formally define MoCCs and implement the solver, an operational semantics was specified for MoCCML. These aspects are described in the Section 3.2.2 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 proposed.

1.2 Perimeter

This document is the version v2.0 (final version) of the D2.2.1 deliverable dedicated to the MoCC Edition Task. The MoCC edition task is part of the *language workbench* activities. During this specific task, the MoCCs are defined and used to provide concurrency models to DSLs (by ECL mapping, see Section 3.2.1). The DSL instance model integrating MoCC properties (e.g. the extendedCCSL model file) can be used in the GEMOC STUDIO for verification and validation. Indeed, the GEMOC STUDIO execution engine can launch the solver that finds possible execution paths and execution steps on the constrained model.

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.
- **DSL**: Domain-Specific Language.
- **DSML**: Domain-Specific (Modeling) Language.
- **Dynamic Semantics**: see *Execution semantics*.
- **Eclipse Plugin**: an Eclipse plugin is a Java project with associated metadata that can be bundled and deployed as a contribution to an Eclipse-based IDE.
- **ED**: Execution Data (part of DSA).

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

1.4 Summary

The document will present the implemented artifacts i.e. the MoCCML editor, its operational semantics and the MoCC solver.

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.

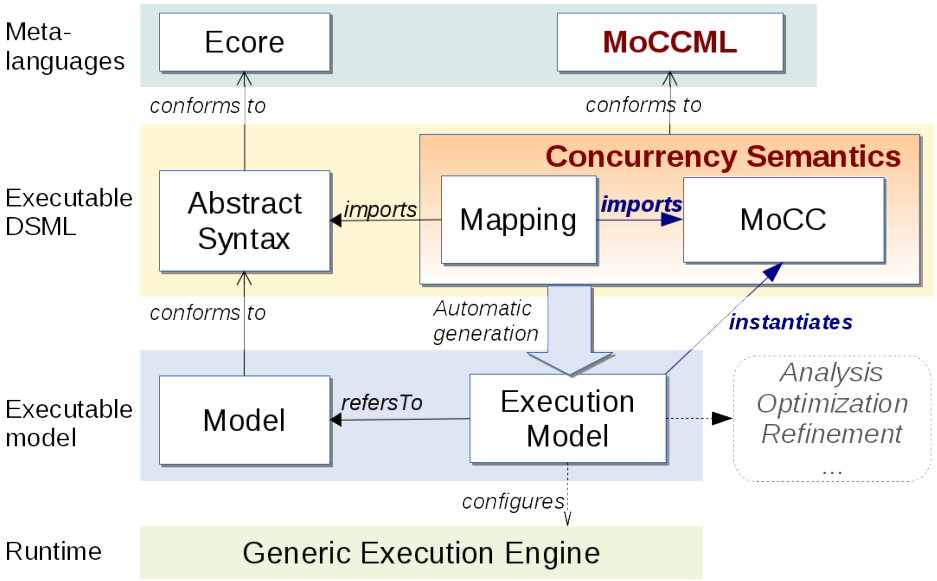

Figure 2.1: MoCCML *Big Picture*

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 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

This section is structured as follows: Section 3.1 briefly reminds the main concepts of the MoCCML abstract syntax[1] and continues by presenting the concrete syntaxes that are proposed to model MoCC; Section 3.2 briefly presents some mandatory steps[2] before the solver is called, then continues with the description of the Operational Semantics of MoCCML and the presentation of the implemented solver extension; Section 3.3 lists the provided plugins that represent the *Software*.

[1] For more details on the abstract syntax, please refer to deliverable D2.1.1
[2] For more details on the ECL mapping, please refer to deliverable D1.3.1

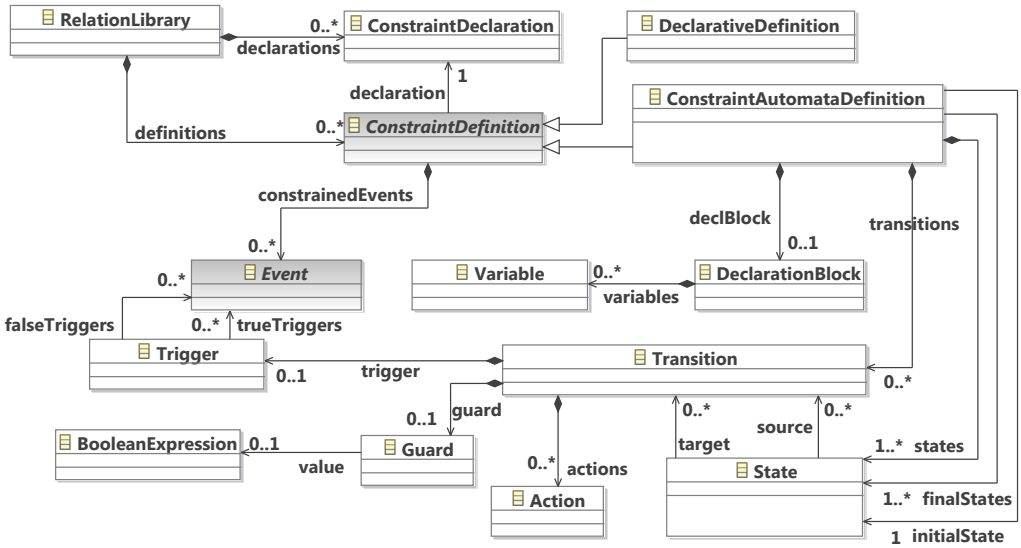

Figure 3.1: Excerpt of MoCCML *Metamodel*

3.1 MoCCML Editor

3.1.1 MoCCML *Abstract Syntax's main concepts*

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 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 MoCCML *Concrete Syntax*

Overview

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 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.

The graphical model shown in Figure 3.2 defines a sample of the kind of editor that we want to provide for the MoCCML concrete syntax. 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 of a *Place*

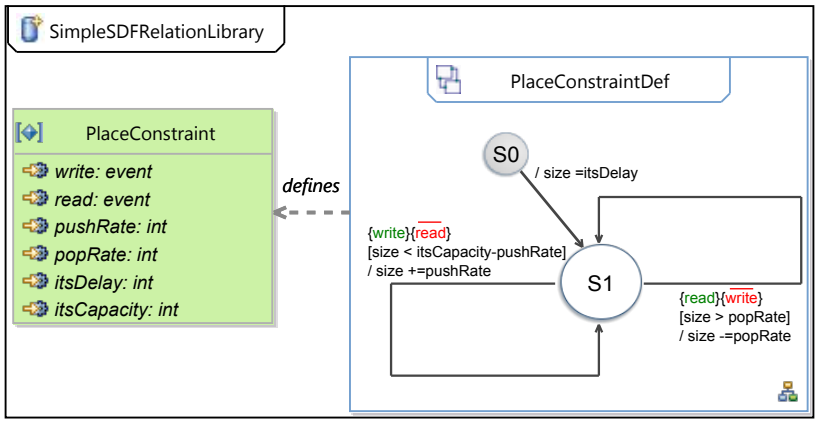

Figure 3.2: Overview of the MoCC *Library Graphical Representation*

implemented as a constrained queue. The automaton operates on 5 integer parameters (one variable: *size*; and 4 constants: *itsCapacity*, *itsDelay*, *pushRate*, *popRate*), which are set during the instantiation process.

Graphical Concrete Syntax

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) illustrated by Figure 3.3; another for the implementation of the relations in the form of automata, illustrated by Figure 3.4.

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 MoCCML Editor implementation proposes different functionalities that can be used during MoCC modeling. As shown in Figure 3.3, to edit a new MoCC library, we propose an *Edition Pallet* containing several subcategories of edition tools. The top level category called *Library Edition* defines the shortcut icons to call a new element or relation instantiation. The types of elements that can be instantiated in this category are new **Relation Library**s, new **Relation Declaration**s, new **Automata Definition**s, new association between an **Automata Definition** and its declaration (Each defined **Relation Declaration** is associated to a **Automata Definition**. The association is provided by **Set Declaration Relation**) and finally the new definition of formal parameters for relation declarations. The last edition category called *Types & Imports* defines shortcuts to instantiate new imports (CCSL library import, CCSL kernel import, etc.) and to instantiate new Primitive types that can be used as local variables type or formal parameters type.

During the MoCC edition process, the edited models cannot be saved at any step during edition. Only the valid models can be saved. The models are valid when they respect the constraints defined by the metamodel (e.g. upper bounds and lower bounds multiplicities). Any invalid model that is saved will cause an error message box stating that: the model cannot be saved, due to metamodel constraint violation.

The edition category in between i.e. *New Library and Required Feature* is an improvement of the top level edition category to ease the editing and saving of instantiated models. It provides some specific MoCC model instantiation features to create **Relation Library**s, **Automata Definition**s and **Relation Declaration**s. This Pallet provides the possibility to define: new **Relation Library**s that contains de-facto a new **Relation Declaration** and one declared formal Parameter; a new **Automata Definition**s that contain de-facto one initial state; new **Relation Declaration**s that contains de-facto one formal Parameter (these are for instance some constraints defined in the MoCCML abstract syntax). The first level of representation doesn't propose the implementation of the automata associated to the **Automata Definition**s. This is the purpose of the second level of representation that will be presented below. The navigation between the two levels is accessible from the **Automata Definition** visualization, on right-click. We can either, create a new automata implementation or open and edit an existing one for each defined **Automata Definition**. 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.

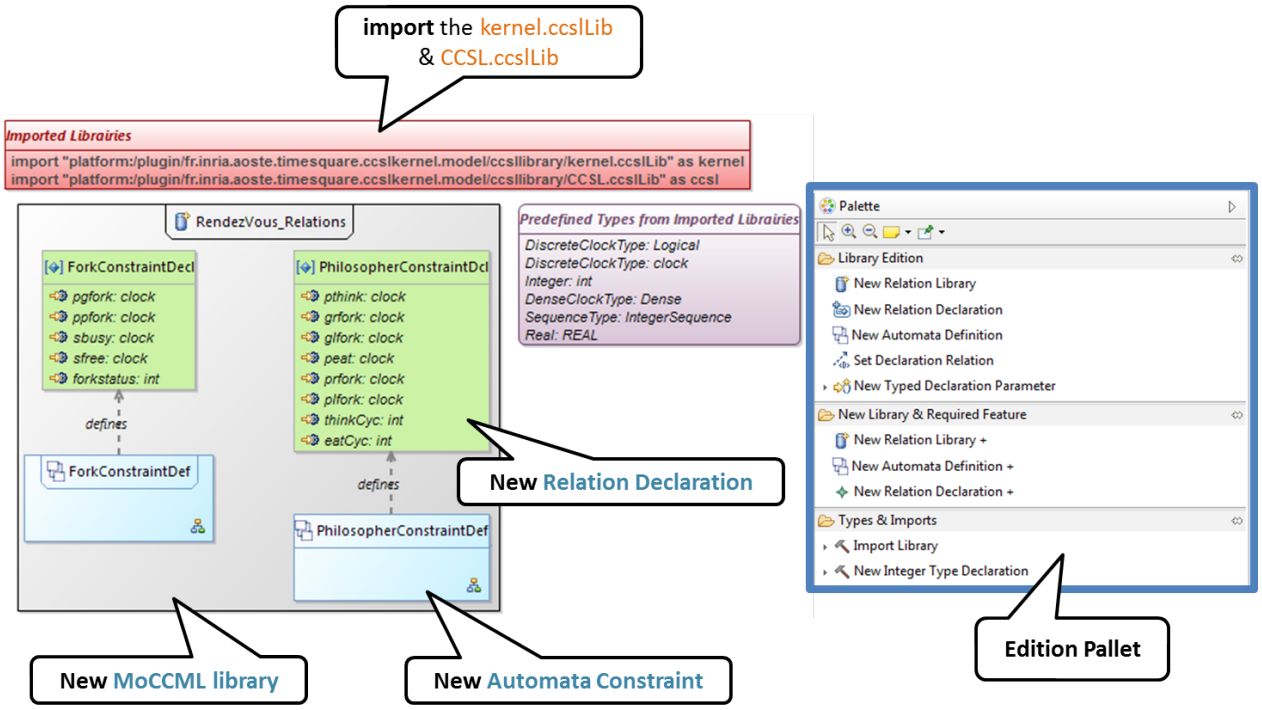

Figure 3.3: First Level of graphical MoCCML *model Representation*

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 *AutomataConstraintDefinition* ie a new *RelationDeclaration*
Declaration Parameter	Syntax used to declare formal parameters for a *RelationDeclaration* eg *AbstractAction*
Automata Definition	Syntax used to declare a *AutomataConstraintDefinition*
Set Declaration Relation	Syntax used to define the association of a *AutomataConstraintDefinition* to its *RelationDeclaration*

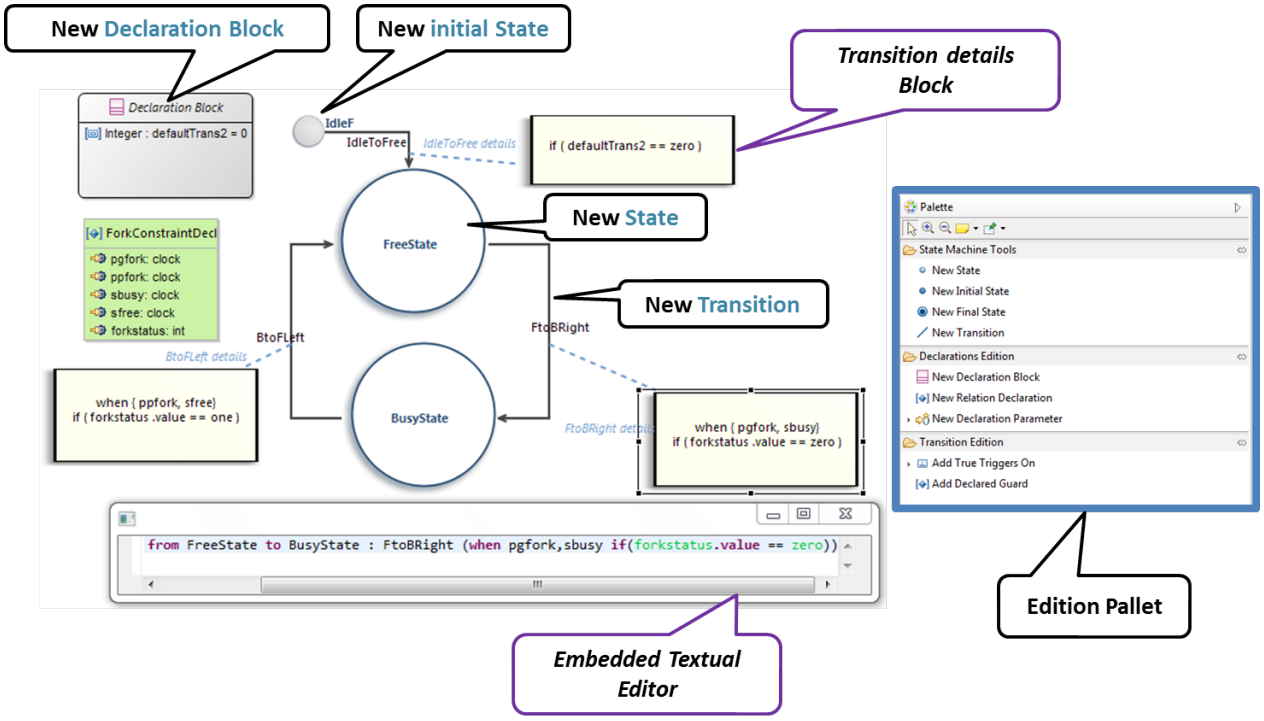

Figure 3.4: Second Level of graphical MoCCML *model Representation*

Table 3.2: Description of the graphical key elements for AutomataConstraintDefinition

AutomataConstraintDefinition 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

In the second level of representation, we also propose an Edition Pallet defining three subcategories of edition tools: *State Machine Tools*, *Declarations Edition* and *Transition Edition*. The first category i.e. *State Machine Tools* defines the shortcut icons to specify new instances of *State*s, *Initial State*, *Final State*s and *Transition*s. The *Declarations Edition* category proposes an icon shortcut to create a set of local variables (New Declaration Block), to add new variables the user must double-click on the displayed *Declaration Block* instance; two shortcuts to create respectively new instances of Relation Declaration and new instances of formal Parameters for the Relation Declarations (these shortcuts are similar to the one defined in the first level of representation). The last category i.e. *Transition Edition* is used to add triggers on the Transitions.

Finally, to complete the edition of automata several graphical elements give access to an embedded textual editor. The graphical elements giving access to the embedded editor are presented in Figure 3.5 and 3.6. The textual editor is launched by double-click on the graphical representation of the concept. For instance, the actions and guards on the transitions are provided in the form of textual concrete syntax and edited using the embedded textual editor. The local variables are also edited using the embedded textual editor. During the edition, it is preferable not to open and edit several embedded editors in parallel, to avoid model synchronization and validation issues.

On the visualization aspect, a layer (to activate, see Section 4) displays a yellow block that shows the details of a transition (trigger, guard and actions).

Table 3.2 summarizes the defined graphical syntax elements.

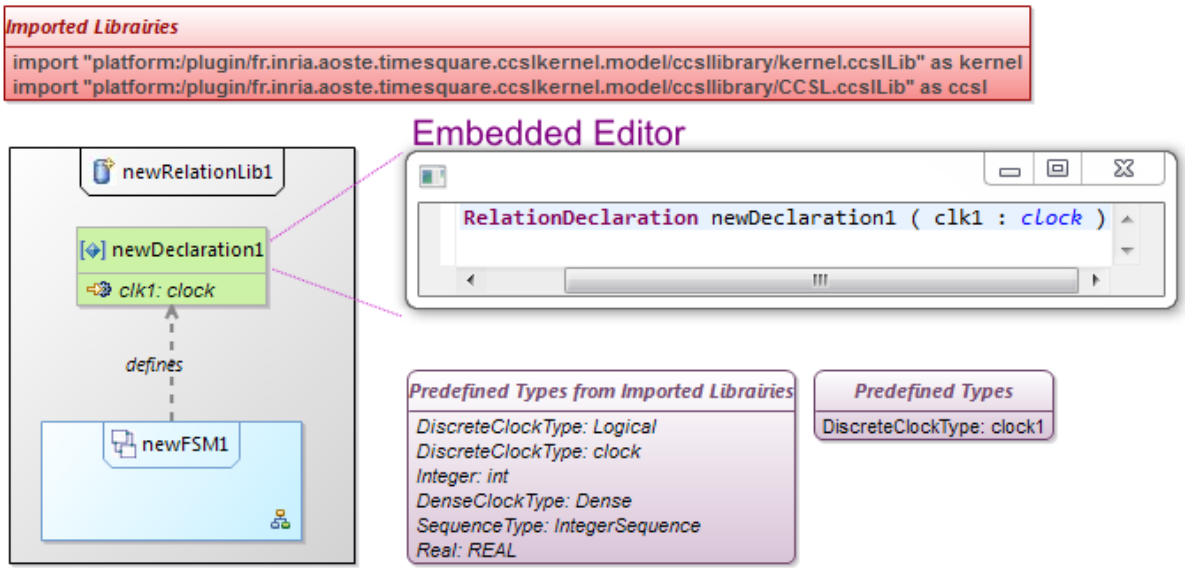

Figure 3.5: Connection point of textual syntax in graphical Representation (Level 1)

In the first level of representation, the connection point is *RelationDeclaration* [3]. In the second level of representation, the connection point is on the *Transition* links and the yellow blocks showing the details of the Transitions.

After validation of the edited models, the mixed graphical/textual model can be serialized as a single textual model. The next Section gives details on the textual concrete syntax of the language.

Textual Concrete Syntax

An xtext grammar is defined for the MoCCML textual concrete syntax. In this grammar, we define the rules to build a MoCCML model with a textual syntax. In each rule, we propose keywords that characterize during edition the nature of the instantiated element. The keywords are summarized in the Table 3.3. The description of the textual concrete syntax is based on the previous automata example in Figure 3.6. 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 *AutomataConstraintLibrary* (*lib*), *RelationLibrary* (*newRelationLib1*), *Automata-ConstraintDefinition* (*newFSM1*).

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

- At line 15, the declaration and initialization of local variables (Integer p = 0, Integer z = 0).
- At line 22, the syntax defines a transition linking two states (from S1 to S3: S1ToS3). Each declared Transition starts with a keyword *from*, followed by the two related states and the name of the transition.The transition has a trigger and a guard ($p>z$). Actions can be defined using a specific syntax that is shown in the Table 3.3.

```
1  AutomataConstraintLibrary lib
2  {
3    import
4    "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/kernel.ccslLib"
5    as kernel ;
6    import
7    "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/CCSL.ccslLib"
8    as ccsl ;
9    DiscreteClockType clock1
10
11   RelationLibrary newRelationLib1 {
```

[3] refers to *ConstraintDeclaration*

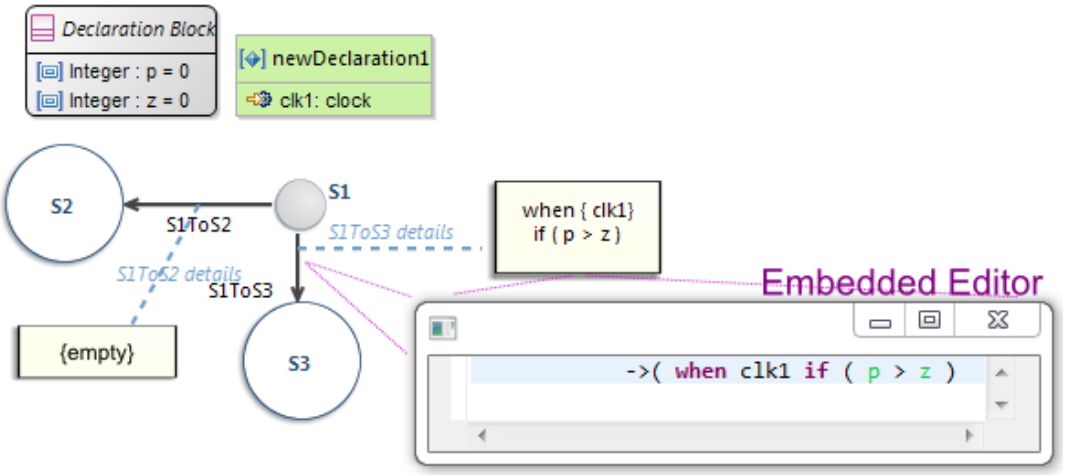

Figure 3.6: Connection point of textual syntax in graphical Representation (Level 2)

```
12
13      AutomataRelationDefinition newFSM1 [ newDeclaration1 ] {
14
15        variables {Integerp = 0Integerz = 0 }
16
17        init: S1
18
19        from S1 to S2 : S1ToS2
20        ->( )
21
22        from S1 to S3 : S1ToS3
23        ->( when clk1 if ( p > z ) do p = ( p + param1.value ) )
24
25        State S1 ( out : S1ToS2, S1ToS3 )
26
27        State S2 ( in : S1ToS2 )
28
29        State S3 ( in : S1ToS3 )
30      }
31      RelationDeclaration newDeclaration1 ( clk1 :clock, param1: int )
32    }
33  }
```

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 (p>z)*; and the action is preceded by the keyword *do* (for example *do p = (p + one)*). The reader can refer to the deliverable D2.1.1 to see the complete grammar of the language.

Table 3.3 summarizes the key elements of the textual syntax.

Some Specificities

In this section, we will explain some specificities of the MoCCML concrete syntax. These specificities are implementation choices to get around some currently unavoidable limitations. Regarding the textual editor, a formatting is defined for the transition such that it focuses on the transition details that are displayed by the textual embedded editor of the graphical editor. When an embedded editor is connected to a given concept of the metamodel, calling the embedded editor from an instance of this concept will show the line/lines corresponding to the instantiated element in a textual form. Trying to modify the code will imply the opening of another window which is to perform the modification.

Table 3.3: Textual Syntax key elements to define a AutomataConstraintLibrary

Textual Syntax	Description
AutomataConstraintLibrary lib	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 newRelationLib1	Syntax used to declare a new automata MoCCML library; (keyword + library name)
RelationDeclaration newDeclaration1 (param1 : *int*, clk1 : *clock*)	Syntax used to declare the interfaces of a *AutomataConstraintDefinition* and their formal parameters
AutomataRelationDefinition newFSM1 $[newDeclaration1]$	Syntax used to declare a new automata MoCCML relation definition; (keyword + library name + interface name)
variables *Integer* p = 0	Syntax used to declare local
init: S1 **finals**: e.g. S4	Syntax used to declare an initial state and final states
from S1 **to** S3 : S1ToS3 (**when** clk1 **if** (p>z) **do** p = (p + one))	Syntax used to declare relation between states (transition), trigger, guard and action for the transition
State S2 (**in**: S1ToS2) or S1 (**out**: S1ToS2) or both *in, out* for a given state.	Syntax to declare for a given state (S1), its input and output transitions

The complete line is always displayed, thus allowing sometimes the edition of model elements that are referenced elsewhere (i.e. edition of cross-referenced elements) and not visible from the opened embedded editor. The automatic update of the cross-referenced model elements that are modified is not currently handled by the textual embedded editor. To solve this issue, we hide the cross-referred elements of the line by introducing line breaks. For instance, when editing a transition, we only want to focus on the edition of the triggers, guard and actions and not on the edition of the states or transition names. Consequently for the transitions, the embedded editor has to be opened on a line that only displays the transition details. We introduce line break formatting on the syntactical element '$\rightarrow$' of the transition, thus only displaying in the embedded editor the detail of a given transition. For instance in Listing 3.1, the line 23 shows the transition details related to the transition S1ToS3. When we call the embedded editor from this transition, only the line 23 is displayed instead of 22 and 23.

Besides, in the MoCCML metamodel the assignments and operations uses a set of defined primitives numerical values to represent the first 10 numbers i.e. 1, 2, 3, ..10. Regarding the graphical syntax, we have already presented the usage of the pallet category *New Library & Required Feature* to ease the instantiation of model and the validation of models. Each instantiated element of this specific pallet contains the required MoCC model elements to save the models. This section is completed in the Tutorial at Section 4.

3.2 MoCCML Executable Model Solver

3.2.1 Prerequisites

Some prerequisites are necessary in the GEMOC STUDIO approach to be actually able to execute and run the instantiated models. It is necessary to at least define the relation between the concept of the DSL and the MoCCML constraint they relates to. This step is done by defining the DSE project and specifying the mappings between the DSL concepts and the MoCC constraints. The mapping file is used along with an instance model of the DSL to generate a complete constrained model that represents all the constraints instances and the relation between constraints in regards to the DSL instance model and the constraint attached to each concept of the instantiated model in GEMOC STUDIO this last model is called time model (*.timemodel).

ECL Mapping

The Listing 3.2 illustrates an example of mapping for a DSL concept called *Agent*.

```
context Agent
  def : start : Event
  def : stop : Event
  def : isExecuting : Event
context Place
 inv PlaceLimitation:
 Relation PlaceConstraint(self.outputPort.write,self.inputPort.read, self.outputPort.rate, self.inputPort.
     rate, self.delay, self.capacity)
```

Listing 3.2: Sample of ECL mapping for a concept Agent

3.2.2 MoCCML *Operational Semantics*

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 a 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* σ 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 on 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 Related plugins in 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*[4]. As a remainder, the *ccslmocc* contains the concepts used to declare MoCC libraries (eg, *AutomataConstraintLibrary*); 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
- 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*

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

- 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 *AutomataConstraintLibrary*. 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. Launching and using MoCCML Model Editor

This chapter gives basic guidelines to create MoCCML models in the GEMOC STUDIO. The MoCC are created using the file extension "*.moccml*".

4.1 MoCCML Project Creation and MoCCML File Creation

To create a new MoCC project, the user can use the MoCCML eclipse wizard. From the eclipse Menu, do File → New → Other → *New MoCCML Project* as shown in Figure 4.1.

Another starting point could be to use the *xDSML project* to create a new MoCC project. From the xDSML editor, the user has access to a link to create new MoCC project. The link is *MoCCML project* as shown in Figure 4.2. Click on the link to display the MoCC project creation wizard.

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 the textual editor.

The file can also be edited in a graphical style using the graphical editor. To define a new graphical model, right-click on the MoCC file and select New → Representations File as shown in Figure 4.3. From the new generated set of file for graphical edition, the root element to create a new diagram can be selected. Right click on the root element of the MoCC and choose the creation of a new Diagram.

4.2 Starting Model Edition

A predefined example of a project implementing a MoCCML model is available in the GIT at *org/gemoc/sample/SigPML_MoCCML/* with a ReadME file describing the steps to use the MoCC files.

The steps to follow are presented by the Figure 4.4 and Figure 4.5. At each step, the Pallets of the graphical editor can be used for graphical edition or the textual editor with the textual concrete syntax.

5. Conclusion and Perspectives

This version of the editor is based on the MoCCML metamodel presented in D2.1.1. The MoCC editor provides a tool to create MoCC definitions in a graphical and textual way. The models can be tested after integration with a DSL and

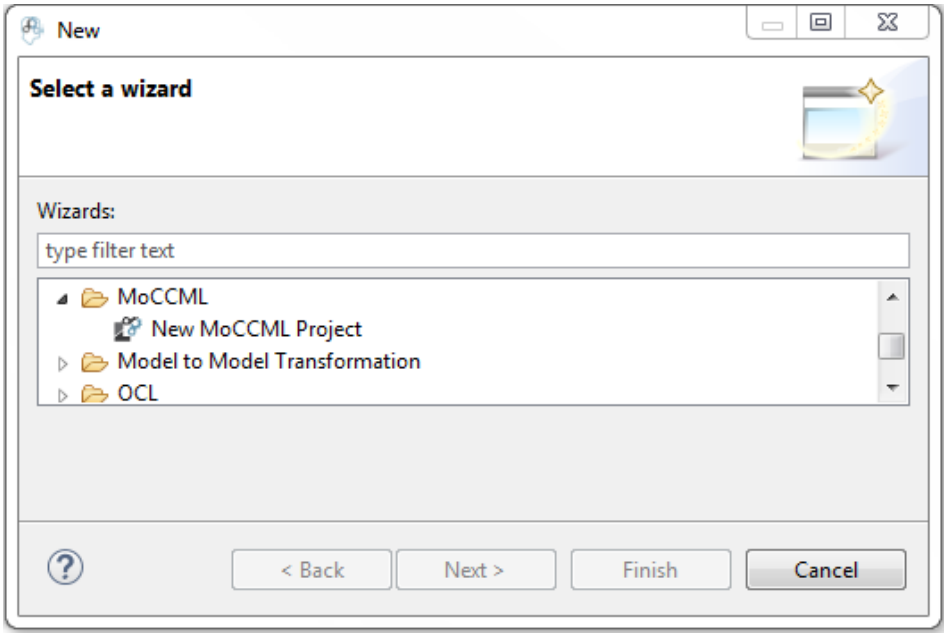

Figure 4.1: Screenshot of a MoCC *Project creation step 1*

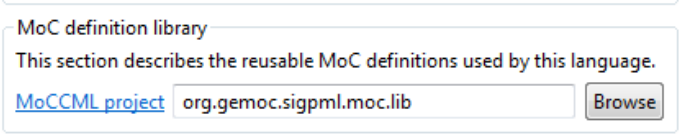

Figure 4.2: Screenshot of a MoCC *Project creation Step 2*

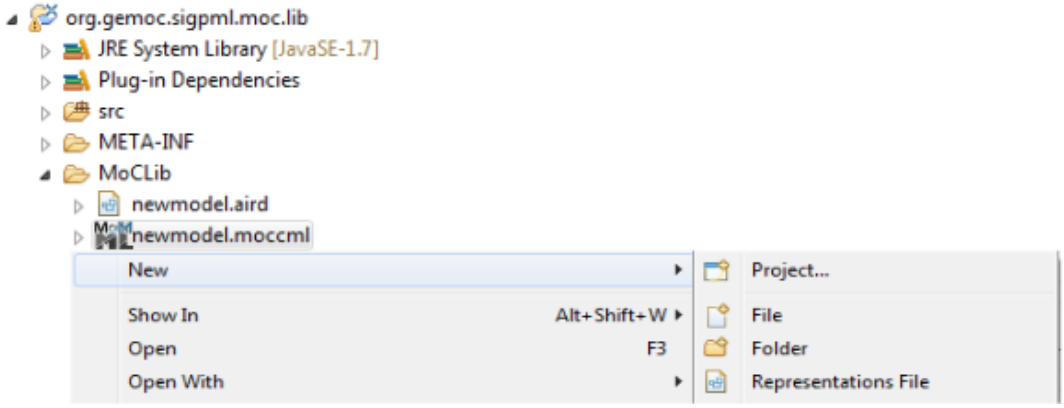

Figure 4.3: Screenshot of a MoCC *New Representation*

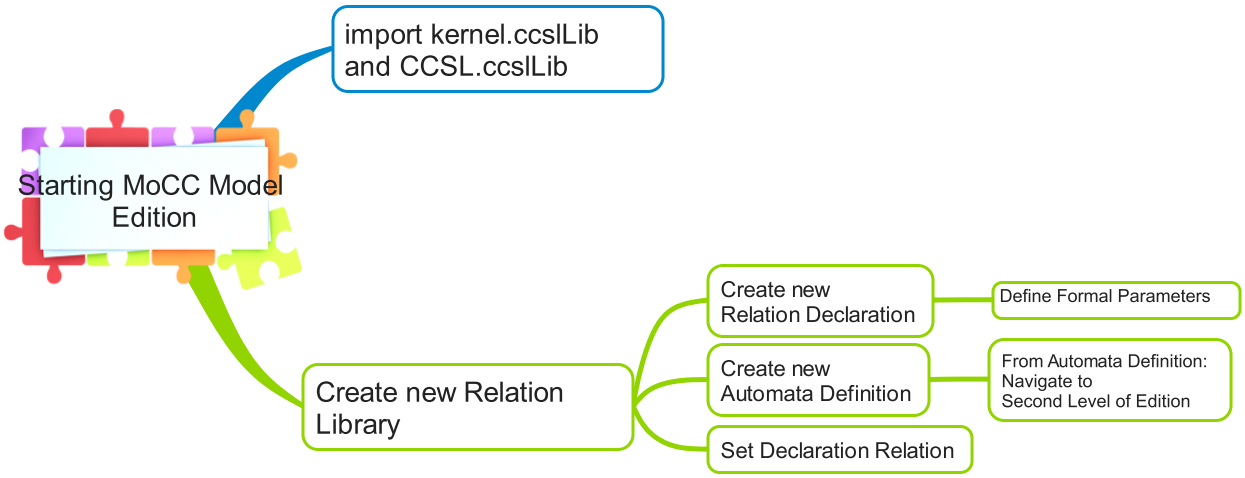

Figure 4.4: Flow to create MoCC

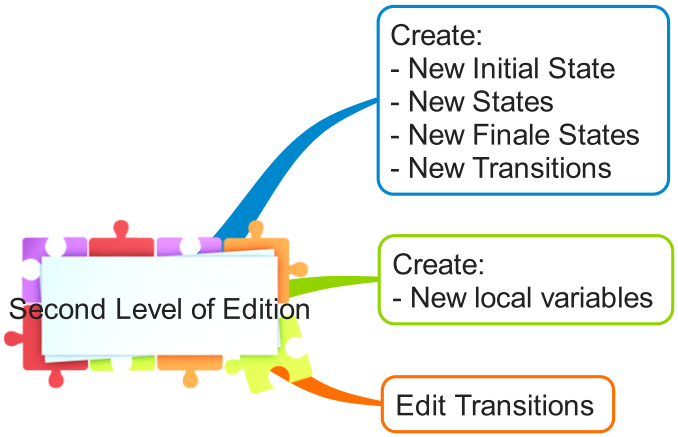

Figure 4.5: Flow to create constraint automata

using the implemented extended solver for TimeSquare. By testing, 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 provided using exhaustive exploration tools (see D 3.4.1). In fact, it could be interesting to have this exhaustive exploration for activities such as model-checking that verifies system properties.

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.

