Operational Semantics of the Model of Concurrency and Communication Language

Julien Deantoni, Papa Issa Diallo, Joël Champeau, Benoit Combemale, Ciprian Teodorov

Operational Semantics of the Model of Concurrency and Communication Language

Julien DeAntoni, Papa Issa Diallo, Joel Champeau, Benoit Combemale, Ciprian Teodorov

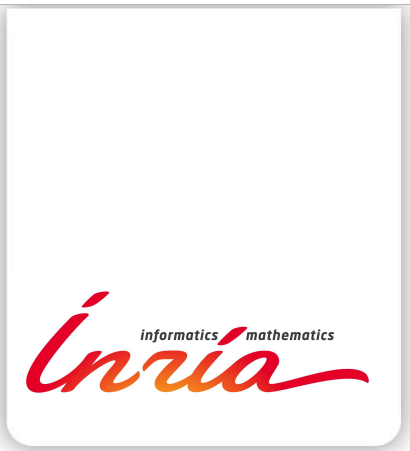
RESEARCH REPORT

N° 8584

August 2014

Project-Teams Aoste and Diverse and STIC-IDM Ensta Bretagne

ISSN 0249-6399 ISRN INRIA/RR--8584--FR+ENG



Operational Semantics of the Model of Concurrency and Communication Language

Julien DeAntoni*, Papa Issa Diallo†, Joel Champeau†, Benoit Combemale ‡, Ciprian Teodorov†

Project-Teams Aoste and Diverse and STIC-IDM Ensta Bretagne

Research Report n° 8584 — August 2014 — 20 pages

Abstract: In the GEMOC project, MoCCML is dedicated to define the concurrency model associated with the DSMLs. The purpose of this document is to define the operational semantics of the MoCCML language and also to define the first steps of an approach to provide an exhaustive exploration of the MoCCML models.
This document presents the operational semantics of the MoCCML language. The chapter is divided in several sections that present the grammar rules of the language and the operational rules mainly defined using mathematical grounds and Plotkin [2] rules. Chapter 3 presents the elements that are relevant to describe the evolution of a MoCCML model. Chapter 4 presents a draft of how exhaustive exploration is to be realized in a context using MoCCML models. Finally Chapter 5 presents the conclusion.

Key-words: Operational semantics, meta language, MoC, language semantics, concurrency

This document was partially founded by the ANR GEMOC project ANR-12-INSE-0011

* University Nice - Sophia Antipolis, CNRS, I3S, UMR 7271
† Ensta Bretagne, CNRS, Lab-STICC, UMR 6285
‡ INRIA Rennes - Bretagne Atlantique

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

MoCCML operational semantics

Résumé : Ce document définit la sémantique opérationnelle du langage MoCCML, un meta langage dédié à la spécification de la concurrence au sein de la définition d'un langage specifique au domaine. Il définit aussi quelques éléments permettant d'aller vers l'exploration exhaustive des modèles MoCCML.

Mots-clés : Sémantique opérationnelle, meta langage, MoC, sémantique, concurrence

Contents

1 Introduction

1.1 Purpose

The MoCCML language is a new formalism to express behavioral semantics of DSLs based on the Model of Computation (MoC) theory. The Metamodel of the language defines concepts to model constraints that control the execution of an application model according to formal execution rules. The modeled constraints are expressed in the form of relations between clocks representing relevant events of the system. As such, the language takes advantage from the CCSL formalism which defines constraints on clocks, and from an operational mechanism of relation description based on finite state automata.

The general idea of adding behavioral semantics to DSLs is driven by the motivation to provide executable models and to make explicit their concurrency model. In order to perform analysis (*e.g.,*by simulation or model-checking), not only the models should be executable, but also their concurrency model should be explicit and formal.

In practice, the executability and concurrency models are covered by the description of the execution semantics of DSMLs, whether it is expressed implicitly or explicitly.

Our purpose in this document is to provide a complete formal semantics for the MoCCML language by using Plotkin's Structural Operational Semantics [2]. This formalism provides a set of rules used to unambiguously specify the behavior or evolution of the elements composing the language. As a result, abstract mathematical reasoning and proof should be applied to the model defined in the language. Moreover, the rules of the formal semantics can be used for the implementation of an execution engine to provide simulation and exhaustive behavior exploration.

There exist different possibilities to write the formal semantics of models *i.e.,*axiomatic semantics, denotational semantics (sometimes referred as transformational semantics), or operational semantics. However, the operational semantics is closer to the definition of simulation engines, as it basically defines the step-by-step execution rules of a given language. The implementation of a simulator for MoCCML models is not the only aspect that operational semantics rules are used for. In fact, a simulator generally shows only the execution traces of one possible execution path. As such, the rules can also be used for the exhaustive exploration of all the possible execution paths.

In this document, we define an operational semantics for the MoCCML language. The sections in the document are structured in such a way that, we firstly provide a description of the grammar rules for the language syntax; then we provide the operational semantics rules. This document also presents two more parts to identify the elements of the MoCCML models that are relevant for simulation and to realize an exhaustive exploration on these elements.

1.2 Perimeter

This document is the version v1 of the D3.2.1 deliverable. The document addresses the operational semantics of the MoCCML language, the description of the elements that are relevant to specify the state of a MoCCML model as well as the rules that specify the evolution of this state. It also gives a first description of the way in which MoCCML models are used for exhaustive exploration.

1.3 Definitions, Acronyms and Abbreviations

- **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.
- **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

In the GEMOC project, MoCCML is dedicated to define the MoC associated with the DSMLs. The purpose of this document is to define the operational semantics of the MoCCML language and also to define the first steps of an approach to provide an exhaustive exploration of the MoCCML models.

Chapter 2 presents the operational semantics of the MoCCML language. The chapter is divided in several sections that present the grammar rules of the language and the operational rules mainly defined using mathematical grounds and Plotkin [2] rules. Chapter 3 presents the elements that are relevant to describe the evolution of a MoCCML model. Chapter 4 presents a draft of how exhaustive exploration is to be realized in a context using MoCCML models. Finally Chapter 5 presents the conclusion.

2 MoCCML semantics

The language is based on the definition of several RelationDefinition which provides clock constraints and the state based definitions. The operational semantics defined in the next sections define the semantics of these RelationDefinition.

First of all, the grammar rules are described using a BNF (Backus-Naur Form) notation. In this notation, we associate a meaning to the following symbols:

- **::=** means *is defined by*
- A | B means parallel composition of A and B (if one of the term A or B is absent it means A or B);

2.1 Operational semantics of the clock constraints

The operational semantics of the clock constraints is the operational semantics of the CCSL language [1].

2.1.1 Syntax of the MoCCML declarative operators

This section defines the syntax of the MoCCML declarative operators on a set of clocks $\mathcal{C}$. With the primitive constructs provided by the kernel, new constructs can be derived and proposed in libraries. For the sake of conciseness, we use a symbolic notation for the kernel operators and constructs. There also exists a concrete textual syntax given in the deliverable D.2.2.1 for the state machine part and on the http://timesquare.inria.fr/index.php?slab=extendedccsl-grammar website for the declarative part.

To ease the writing of the semantics, we use a symbolic notation for the MoCCML declarative operators. The syntax in given in Tables 1 and 3. The relation named "clock definition" is a simple clock relation which associates a clock with a clock expression. $\mathcal{C}$ includes all the clocks in the specification. Notice that in the declarative part of MoCCML, a clock relation applies to two clock references, themselves possibly defined by a clock expression.

Table 1: MoCCML constraints and relations

CC	::=		(clock constraint)
		CC | CR	(parallel composition)
	|	CR	
	|	CR if bool	(conditional constraint)
CR	::=		(clock relation)
		clock r_{op} clock	
	|	clock $\triangleq$ CE	(clock definition)

Continued on next page

Table 2: Relation operators

r_{op}	::=	(relation operator)
	$\subset$	(subclocking)
|	$\#$	(exclusion)
|	$=$	(coincidence)
|	$\prec$	(s_precedence)
|	$\preccurlyeq$	(precedence)

Table 3: MoCCML clock expressions

CE	::=		(clock expression)
		bool ? clock : clock	(conditional expression)
	|	clock	(clock reference)
	|	clock^natural	(wait)
	|	clock $\downdownarrows$ clock	(s_sample)
	|	clock $\downarrow$ clock	(sample)
	|	clock $\lightning$ clock	(upto)
	|	clock $\bullet$ clock	(concat)
	|	clock $+$ clock	(union)
	|	clock $*$ clock	(inter)
	|	clock(naturalSequence) $\rightsquigarrow$ clock	(defer)
	|	clock $\vee$ clock	(sup)
	|	clock $\wedge$ clock	(inf)

2.1.2 Semantics

We propose to give MoCCML an operational structural semantics that allows the effective construction of temporal evolutions. Note that we consider only clocks with a discrete set of instants.

MoCCML Declarative Model A MoCCML *Declarative Model* $\mathcal{M} = \langle \mathcal{C}, \mathcal{S} \rangle$ consists of a finite set of discrete clocks $\mathcal{C}$, constrained by a MoCCML$_{\mathcal{C}}$ specification $\mathcal{S}$. A MoCCML model is a set of constraints specified by a natural extension of CCSL [1]. Consequently, this document is a major update and extension of the [1] document.

MoCC Specification A MoCCML system denotes a set of schedules. If empty, there is no solution, the specification is invalid. If there are many possible schedules, it leaves some freedom to make some choices depending on additional criteria. For instance, some may want to run everything as soon as possible (ASAP), others may want to optimize the usage of resources (*e.g.*,processors/memory/bandwidth).

A *schedule* σ over set of clocks $\mathcal{C}$ is a possibly infinite sequence of ticking clocks. $\sigma : \mathbb{N} \rightarrow 2^{\mathcal{C}}$.

Given a clock c, a *step* $s \in \mathbb{N}$ and a schedule σ, $c \in \sigma(s)$ means that clock c ticks at step s for this particular schedule. f_s^σ denotes the set of ticking clock at step s in σ. The goal of the semantics rules is to be specify how to construct a specific schedule. To ease the reading, in the following rules we denote by F the set of fired clock at the current step for the schedule under construction.

Based on this, we can define the notion of *History*. Given a schedule σ, the *History* over a set of clocks C is a function $H_\sigma : \mathcal{C} \times \mathbb{N} \to \mathbb{N}$ defined inductively as follows for all clocks $c \in C$:

$$H_\sigma(c, 0) = 0 \tag{1}$$

$$\forall n \in \mathbb{N}, c \notin \sigma(n) \implies H_\sigma(c, n+1) = H_\sigma(c, n) \tag{2}$$

$$\forall n \in \mathbb{N}, c \in \sigma(n) \implies H_\sigma(c, n+1) = H_\sigma(c, n) + 1 \tag{3}$$

In consequence, the history of a clock c at the current step during the construction of a schedule σ is $H_\sigma(c, now)$. Also, as a syntactic sugar construction, one can obtain the history in between two step $s1$ and $s2$ by using this construction: $H_\sigma(c, s1, s2)$, which is defined by $H_\sigma(c, s2) - H_\sigma(c, s1)$.

The semantics of a specification $\mathcal{S}$ expressed in MoCCML$_\mathcal{C}$ is given as a Boolean expression on $\mathscr{C}$, where $\mathscr{C}$ is a set of Boolean variables in bijection with $\mathcal{C}$ as defined later in equations 7 and 8.

$$\text{let } \pi : \mathcal{C} \to \mathscr{C} \text{ bijection, and } [\![\ ]\!] : \text{KCCL}_\mathcal{C} \to \mathscr{B}_\mathscr{C} \tag{4}$$

$[\![\ ]\!]$ is defined by structural rewriting rules. For convenience, we denote $\pi(c)$ by c, for all c in $\mathcal{C}$. $\mathsf{c} = 1$ means that $c \in \sigma(now)$. More generally,

$$\begin{aligned} \sigma \models \langle \mathcal{C}, \mathcal{S} \rangle \; iff \\ (\forall f \in \sigma : \mathscr{C} \to \{0,1\}) \, \langle \mathcal{C}, \mathcal{S} \rangle \xrightarrow{f} \Leftrightarrow [\![\mathcal{S}]\!](f) = 1 \end{aligned} \tag{5}$$

In Eq. 5, f is a valuation of $\mathscr{C}$, $[\![\mathcal{S}]\!](f) = 1$ says that $[\![\mathcal{S}]\!]$ evaluates to 1 for the valuation f. In the operational semantics rewriting rules, we refer to F as the subset of $\mathscr{C}$ in which only the clock that ticks at the current step are kept.

The next two subsections detail structural transformations from MoCCML$_\mathcal{C}$ to Boolean expressions on $\mathscr{C}$ (*i.e.,*$\mathscr{B}_\mathscr{C}$). In Boolean expressions, we use operators $\Rightarrow$ (implication), $=$ (equality), $\#$ (exclusion), and ite(, ,) (if ... then ... else ...) such that for any Boolean expression t_1, t_2, t_3:

$$\begin{aligned} t_1 \Rightarrow t_2 &\Leftrightarrow \neg t_1 \vee t_2 \\ t_1 = t_2 &\Leftrightarrow (t_1 \wedge t_2) \vee (\neg t_1 \wedge \neg t_2) \\ t_1 \# t_2 &\Leftrightarrow \neg t_1 \vee \neg t_2 \\ \text{ite}(t_1, t_2, t_3) &\Leftrightarrow (t_1 \wedge t_2) \vee (\neg t_1 \wedge t_3) \end{aligned}$$

A first rule is given right now. This rule expresses the composition of clock relations: the parallel composition of clock relations is the conjunction of the associated Boolean expressions.

$$[\![\mathsf{CR}_1 \mid \mathsf{CR}_2]\!] = [\![\mathsf{CR}_1]\!] \wedge [\![\mathsf{CR}_2]\!] \quad (\text{ paral }) \tag{6}$$

Clocks A clock c is a possibly infinite ordered set of instant $\mathcal{I}_c$. $\mathcal{I}_c$ possibly contains two specific instants, its birth denoted $c^\dagger$ and its death denoted $c^\downarrow$. The birth always precedes the first instant of its clock and its death is its last instant (possibly synchronous with another instant).

$$\frac{c^\dagger \in \mathcal{I}_c \wedge c^\downarrow \notin \mathcal{I}_c}{[\![c]\!] = \mathsf{c}} \quad (\text{ clock1 }) \tag{7}$$

$$\frac{c^\dagger \notin \mathcal{I}_c \vee c^\downarrow \in \mathcal{I}_c}{[\![c]\!] = \neg\mathsf{c}} \quad (\text{ clock2 }) \tag{8}$$

2.1.3 Clock relations

for some of the following rules, we can distinguish two different natures for the semantics rules. The first kind defines the projection of the constraint to their Boolean representation. The second kind defines the possible propagation of birth and death among the parameter of the constraints.

Conditional clock relation A Clock relation can be defined conditionally to some Boolean β. When β is false, $[\![\mathsf{CR}\text{ if }\beta]\!]$ is true, whatever the clock relation. Else, we have to compute the Boolean expression associated with the clock relation.

$$[\![\mathsf{CR}\text{ if }\beta]\!] = (\beta \Rightarrow [\![\mathsf{CR}]\!]) \quad (\text{ rcond }) \tag{9}$$

History-independent clock relations *Sub-clocking* c_1 is a subclock of c_2 (or c_2 is a superclock of c_1) means that each instant of c_1 must be coincident with an instant of c_2. In logical words this says that c_1 ticks only if c_2 ticks, hence the logical implication.

$$[\![c_1 \boxed{\subset} c_2]\!] = (\mathsf{c}_1 \Rightarrow \mathsf{c}_2) \quad (\text{ subclock }) \tag{10}$$

$$c_2^\downarrow \Rightarrow c_1^\downarrow \quad (\text{ subclock death propagation }) \tag{11}$$

Recall that $\mathsf{c} = [\![c]\!]$.

Clock exclusion Two clocks c_1 and c_2 can be declared exclusive, that is, none of their instants are coincident, or equivalently, it is forbidden that both c_1 and c_2 tick at a configuration. This is expressed by the Boolean expression $\mathsf{c}_1 \# \mathsf{c}_2$ equivalent to $\neg(\mathsf{c}_1 \wedge \mathsf{c}_2)$ and $\neg\mathsf{c}_1 \vee \neg\mathsf{c}_2$.

$$\left[\!\!\left[c_1 \boxed{\#} c_2\right]\!\!\right] = (\mathsf{c}_1 \# \mathsf{c}_2) \quad (\text{ excl }) \tag{12}$$

Clock equality This is a special case of double subclocking, there is a bijection between the sets of instants of the two clocks. The Boolean expression states that c_1 ticks if and only if c_2 ticks and conversely.

$$[\![c_1 \boxed{=} c_2]\!] = (\mathsf{c}_1 = \mathsf{c}_2) \quad (\text{ coinc }) \tag{13}$$

$$c_2^{\downarrow} \Leftrightarrow c_1^{\downarrow} \quad \text{(coinc death propagation)} \tag{14}$$

Clock definition The left-hand side clock ticks whenever the right-hand side clock expression ticks; also the death of one of them is propagated to the other one.

$$\left[\!\left[c \boxed{\triangleq} \mathsf{CE} \right]\!\right] = (\mathsf{c} = [\![\mathsf{CE}]\!]) \quad \text{(clockDefinition)} \tag{15}$$

$$c^{\downarrow} \Leftrightarrow [\![\mathsf{CE}]\!]^{\downarrow} \quad \text{(clockDefinition death propagation)} \tag{16}$$

History-dependent clock relations The next two clock relations depend on the history of the concerned clocks. More precisely they depend on the difference of their own history. Let $\delta \triangleq H(c_1, now) - H(c_2, now)$.

Clock precedence $c_1 \boxed{\prec} c_2$ is read "c_1 precedes c_2". This means that for any step s in a schedule that satisfies the MoCCML specification, $H(c_1, s) \geq H(c_2, s)$. This formulation is less intuitive than the following: for any natural number k, the k^{th} instant of c_1 strictly precedes the k^{th} instant of c_2. This precedence between instants explains that this relation is also read as "c_1 is *faster than* c_2". According to this definition, c_1, which is the faster of the two clocks, is never constrained. As for c_2, it is constrained only when its index becomes equal to the index of c_1. Under such a circonstance, c_2 can not tick.

$$\frac{\beta \triangleq (\delta = 0)}{[\![c_1 \boxed{\prec} c_2]\!] = (\beta \Rightarrow \neg \mathsf{c}_2)} \quad \text{(precede)} \tag{17}$$

Another consequence of this rule is that the following invariant property holds:
Invariant: $\delta \geqslant 0$

$$\frac{\begin{array}{c} c_1^{\downarrow} \in \mathcal{I}_{c_1} \wedge \\ c_2^{\downarrow} \notin \mathcal{I}_{c_2} \wedge \\ \delta = 0 \end{array}}{c_1^{\downarrow} \Rightarrow c_2^{\downarrow}} \quad \text{(precede death propagation)} \tag{18}$$

Clock causality The cause relation is similar to the previous one. The unique difference is in the possibility for c_2 to tick when $\delta = 0$, provided that c_1 also ticks. Hence the Boolean expression involves two implications.

$$\frac{\beta \triangleq (\delta = 0)}{\left[\!\left[c_1 \boxed{\preccurlyeq} c_2 \right]\!\right] = \big(\beta \Rightarrow (\mathsf{c}_2 \Rightarrow \mathsf{c}_1)\big)} \quad \text{(prec)} \tag{19}$$

The invariant property on δ still holds:
Invariant: $\delta \geqslant 0$

$$\frac{\begin{array}{c} c_1^{\downarrow} \in \mathcal{I}_{c_1} \wedge \\ c_2^{\downarrow} \notin \mathcal{I}_{c_2} \wedge \\ \delta = 0 \end{array}}{c_1^{\downarrow} \Rightarrow c_2^{\downarrow}} \quad \text{(cause death propagation)} \tag{20}$$

2.1.4 Clock expressions

During the construction of a schedule, clock expressions may change. So, we introduce *conditional rewriting rules* for clock expressions. A rewriting is expressed as $\mathsf{CE} \rightarrow \mathsf{CE}'$ where CE' replaces CE after a firing which meets the condition. Of course, the constraint parameters can also be rewritten.

A clock expression has an associated *implicit clock* In the rules below, c stands for the clock associated with the current clock expression.

For all the following semantic rules; there is no death propagation since it can always be added by using additional relations.

Conditional clock expression A conditional clock expression defines a clock that behaves either as a clock c_1 or as another clock c_2 according to the value taken by the Boolean β.

$$[\![\beta \ ? \ c_1 : c_2]\!] = \text{ite}(\beta, \mathsf{c}_1, \mathsf{c}_2) \quad (\text{ econd }) \tag{21}$$

Terminating clock expressions Terminating clock expressions define finite clocks (*i.e.,*clocks that eventually die). These clock expressions are used to build more complex clock expressions, especially through the clock concatenation.

Wait The wait clock expression $c1\hat{}n$ ticks in coincidence with the next n^{th} strictly future tick of c_1, and then dies.

$$\frac{\beta \triangleq (n = 1)}{[\![c_1\hat{}n]\!] = (\beta \wedge \mathsf{c}_1)} \quad (\text{MoCCML await }) \tag{22}$$

$$\frac{c_1 \in F}{c_1\hat{}1 \rightarrow \mathcal{I}_c = \mathcal{I}_c + c^{\downarrow}} \quad (\text{ RWawait1 }) \tag{23}$$

$$\frac{c_1 \in F \quad n > 1}{c_1\hat{}n \rightarrow c_1\hat{}(n-1)} \quad (\text{ RWawait2 }) \tag{24}$$

Strict sampling Sampling clock expressions involve two clocks. The first is considered as a trigger and the second as a time base. The sampling expression ticks in coincidence with the tick of the base clock immediately following a tick of the trigger clock, and then dies. There exist two versions of the sampling: either the strict one (the coincident tick of the base clock is strictly after the trigger tick) or the non-strict one (the coincident tick of the base clock may be coincident with the trigger tick when this one is coincident with a base clock tick).

$$[\![c_1 \Downarrow c_2]\!] = c \quad (\text{ ssampl }) \tag{25}$$

$$\frac{c_1 \in F}{c_1 \Downarrow c_2 \rightarrow c_2\hat{}1} \quad (\text{ RWssampl }) \tag{26}$$

Non strict sampling

$$[\![c_1 \downarrow c_2]\!] = (\mathsf{c}_1 \wedge \mathsf{c}_2) \quad (\text{ sampl }) \tag{27}$$

$$\frac{c_1 \in F \quad c_2 \in F}{c_1 \downarrow c_2 \rightarrow \mathcal{I}_c = \mathcal{I}_c + c^{\downarrow}_{\dagger}} \quad (\text{ RWsampl1 }) \tag{28}$$

$$\frac{c_1 \in F \quad c_2 \notin F}{c_1 \downarrow c_2 \rightarrow c_2\hat{}1} \quad (\text{ RWsampl2 }) \tag{29}$$

Preemption The premption expression $c_1 \not\downarrow c_2$ behaves like c_1 while c_2 does not tick. When c_2 ticks, the expression dies.

$$[\![c_1 \not\downarrow c_2]\!] = (\mathsf{c}_1 \wedge \neg \mathsf{c}_2) \quad (\text{ upto }) \tag{30}$$

$$\frac{c_2 \in F}{c_1 \not\downarrow c_2 \rightarrow \mathcal{I}_c = \mathcal{I}_c + c^{\downarrow}_{\dagger}} \quad (\text{ RWupto }) \tag{31}$$

Non-terminating index-independent clock expressions These clock expressions contrast with the terminating ones: they don't have explicit death. Among them, many are history-independent.

Clock concatenation The concatenation clock expression $c_1 \bullet c_2$ behaves like c_1 up to the death of c_1. When c_1 dies, the expression behaves like c_2. The concatenation may induce recursive definitions.

$$[\![c_1 \bullet c_2]\!] = \mathsf{c}_1 \quad (\text{ concat }) \tag{32}$$

$$\frac{\begin{array}{c} c \neq c_2 \\ c_1^{\downarrow}_{\dagger} \in \mathcal{I}_{c_1} \end{array}}{c_1 \bullet c_2 \rightarrow c_2} \quad (\text{ RWconcat }) \tag{33}$$

$$\frac{c_1^{\downarrow}_{\dagger} \in \mathcal{I}_{c_1}}{c_1 \bullet c \rightarrow \mathcal{I}_{c_1} = \mathcal{I}_{c_1} - {}_c1^{\downarrow}_{\dagger}} \quad (\text{ RWrecur }) \tag{34}$$

Clock union The union clock expression $c_1 + c_2$ ticks whenever c_1 or c_2 ticks.

$$[\![c_1 + c_2]\!] = (\mathsf{c}_1 \vee \mathsf{c}_2) \quad (\text{ union }) \tag{35}$$

Clock intersection The intersection clock expression $c_1 * c_2$ ticks whenever both c_1 and c_2 tick.

$$[\![c_1 * c_2]\!] = (\mathsf{c}_1 \wedge \mathsf{c}_2) \quad (\text{ inter }) \tag{36}$$

Clock delay The delay clock expression $c_1(ns) \rightsquigarrow c_2$ is a rather complex expression involving two clocks (c_1, c_2) and a sequence of natural numbers (ns). c_1 is a trigger, c_2 a base clock. At each tick of c_1 the head of ns is dequeued and encoded in a binary word bw associated with the expression. This binary word is a kind of "diary" that contains the future rendez-vous with c_2 ticks. For instance, when c_1 ticks and the head of ns is 5, then the expression is expected to tick in coincidence with the next 5^{th} tick of c_2. Note that rendez-vous are not necessarily taken in a monotonic increasing order.

$$\frac{\beta \triangleq (bw = 1.w)}{[\![c_1(ns) \rightsquigarrow c_2]\!] = (\beta \wedge \mathsf{c}_2)} \quad (\text{ defer }) \tag{37}$$

$$\frac{c_1 \notin F \quad c_2 \in F \quad b \in \{0,1\}}{c_1(ns) \rightsquigarrow c_2, b.w \rightarrow c_1(ns) \rightsquigarrow c_2, w} \quad (\text{ RWdefer1 }) \tag{38}$$

$$\frac{c_1 \in F \quad c_2 \notin F \quad h \in \mathbb{N}^\star}{c_1(h.s) \rightsquigarrow c_2, w \rightarrow c_1(s) \rightsquigarrow c_2, w + (0^{h-1}.1)} \quad (\text{ RWdefer2 }) \tag{39}$$

$$\frac{c_1 \in F \quad c_2 \in F \quad b \in \{0,1\} \quad h \in \mathbb{N}^\star}{c_1(h.s) \rightsquigarrow c_2, b.w \rightarrow c_1(s) \rightsquigarrow c_2, w + (0^{h-1}.1)} \quad (\text{ RWdefer3 }) \tag{40}$$

Non-terminating index-dependent clock expressions The last two clock expressions depend on the clock history H, or more precisely on the difference of two clocks histories.

The fastest of slower clocks The sup clock expression $c_1 \vee c_2$ defines a clock that is slower than both c_1 and c_2 and whose k^{th} tick is coincident with the later of the k^{th} tick of c_1 and c_2.

$$\frac{\begin{array}{c}\beta_1 \triangleq (H(c_1) = H(c)) \\ \beta_2 \triangleq (H(c_2) = H(c))\end{array}}{[\![c_1 \vee c_2]\!] = \big((\beta_1 \Rightarrow c_1) \wedge (\beta_2 \Rightarrow c_2)\big)} \quad (\text{ sup }) \tag{41}$$

Invariant: $H(c) = \min\{H(c_1), H(c_2)\}$ and $\beta_1 \vee \beta_2 = 1$

The slowest of faster clocks This expression is the dual of the previous one. The inf clock expression $c_1 \wedge c_2$ defines a clock that is faster than both c_1 and c_2 and whose k^{th} tick is coincident with the earlier of the k^{th} tick of c_1 and c_2.

$$\frac{\begin{array}{c}\beta_1 \triangleq (H(c) = H(c_1)) \\ \beta_2 \triangleq (H(c) = H(c_2))\end{array}}{[\![c_1 \wedge c_2]\!] = \big((\beta_1 \wedge c_1) \vee (\beta_2 \wedge c_2)\big)} \quad (\text{ inf }) \tag{42}$$

Invariant: $H(c) = \max\{H(c_1), H(c_2)\}$ and $\beta_1 \vee \beta_2 = 1$

2.2 State machine operational semantics

The MoCCML language allows the definition of libraries of constraints on clocks with equational and state based support. The libraries are defined throught *RelationDefinition* and *StateBasedRelationDefinition*. As such, we assume that the elements defined in the *StateRelationBasedLibrary* are equivalent to clock relations. For instance, the *StateBasedRelationDefinition* is a primitive for the description of a new relation between clocks. The description of the semantics is divided in two parts: The first part gives a description of the grammar rules highlighting the syntax of the MoCCML; the second part defines the dynamic evolution of state-based clock relations defined with MoCCML.

2.2.1 Syntax Notation

For the sack of simplicity, we will consider that a CR_{fsm} is defined between several clocks, sometimes using additional formal parameters that are not clocks, eg Integer, Real or Sequence. A CR_{fsm} can also be defined as a set of clocks *assiociated with* ($\triangleq$) an CE (Clock Definition). Table 4 shows the rules corresponding to these considerations.

Table 4: State-Based Relation grammar Notation

CR_{fsm} ::=	$(clock)^*\ r_{op}^{fsm}\ (clock)^*$	(simple state-based clock relation)
	$\mid (clock)^* \triangleq CE$	(clock definition)

r_{op}^{fsm} is a special type of clock relation operator that defines constraints between clocks using finite state machine (FSM). A *clock* is a formal parameter of the clock relation, and is defined as a discrete clock type. The set of all *clock*s is defined by $\mathcal{C}$. The formal parameters that are different from clocks (eg *Integer* or *Real*) are used for the evaluation of guards or for actions within the transitions.

The impact of contraints on behaviors is dependent of the evaluation of guards and triggers (presence, absence, returned evalutation value). Accordingly, the grammar rules of the *StateBasedRelationDefinition*s are presented in the Tables 5 and 6 that declare the primitives composing the *StateBasedRelationDefinition*.

Table 5: Grammar rules for MoCCML *StateBasedRelationDefinition*- Part 1

r_{op}^{fsm}	::=	$(DB)?\ (Q \mid T)*$	(fsm-based clock relation operator)
DB	::=	$(LE \mid BEI \mid BE \mid C_E)*$	(declaration block)
Q	::=	$Q_0\ Q_I\ Q_T$	(state set)
Q_0	::=	*state*	(initial state)
Q_I	::=	$(state)*$	(intermediate state set)
Q_T	::=	$(state)*$	(terminal state set)
T	::=	$Q \rightarrow (G)?\ /\ (A)? \rightarrow Q$	(transition)

Naturally speaking, a *StateBasedRelationDefinition* is composed of declaration blocks DB, of a set of states Q and set of transitions T. The declaration blocks is made up with linear expressions LE on Integers, boolean expressions on Integers BEI, classical boolean expressions

Table 6: Grammar rules for MoCCML *StateBasedRelationDefinition*- Part 2

C_E	::=	*clock* | *int*	(set of concrete entities)
LE	::=	*int* LE_{op} *int*	(linear expression)
LE_{op}	::=	+ | -	(linear expression operator)
BEI	::=	*int* BEI_{op} *int*	(boolean expression on integers)
BEI_{op}	::=	= | < | >	(operators on integer boolean expressions)
BE	::=		(boolean expression)
		BEI | BE or BE | BE and BE | not BE	
G	::=	$(BE)?$ $(C_{True})?$ $(C_{False})?$	(guard)
C_{True}	::=	$(clock)*$	(true trigger)
C_{False}	::=	$(clock)*$	(false trigger)
A	::=	$(A_{assign} \mid A_{block} \mid A_{finish})*$	(action)
A_{Assign}	::=	$int = LE$	(integer assignment action)
A_{bloc}	::=	$(A_{assign})+$	(integer assignment block action)
A_{finish}	::=	kill(*clock*)	(finish clock action)

BE or concrete entities C_E, *i.e.,clock* or *int*. The set of states is made up with an initial state $Q_0 \in Q$, several intermediate states ($Q_I \in Q \ \wedge Q_0 \notin Q_I \wedge Q_T \cap Q_I = \emptyset$) and possibly several finale (or terminal) states ($Q_T \in Q \wedge Q_0 \notin Q_T \wedge Q_T \cap Q_I = \emptyset$). States can have input or ouput transitions except for the finale states that do not define output transitions. A transition goes from a state to another and can possibly define a guard and an action. A guard contains from 0 to 3 parts: an optional boolean guard, a first set of clock, which represent the clock that need to tick for the transition to be triggered and a second set of clock, which represent the clock that need to not tick for the transition to be triggered. Finally, there are three different kind of actions (*i.e.,IntegerAssignment*, *IntegerAssignmentBlock* or *FinishClock*).

2.2.2 State based relation operational Semantics

As described in Section 2.1.2, a MoCCML system is a tuple $\langle \mathcal{C}, \mathcal{S} \rangle, H$, where $\mathcal{C}$ is the set of Clock, $\mathcal{S}$ is the specification of clock constraints and H is a given history.

The semantics $[\![r_{op}^{fsm}]\!]$ of r_{op}^{fsm} is given as a Boolean expression on $\mathscr{C}$, where $\mathscr{C}$ is a set of Boolean variables in bijection with $\mathcal{C}$ which is obtained according to a given history, the current state cr_{st} of r_{op}^{fsm}, where $cr_{st} \in Q$; the evaluation of boolean guards and of the triggers on transitions of a *StateBasedRelationDefinition*. The rewriting rules depends on the fired transition and their actions.

If we consider the Figure 1 illustrating an example of state-transition automata. We define cr_{st} as the current state of the state machine where $cr_{st} \in Q$. For a given cr_{st} (crt) which is not the initial state, there is possibly one to many possible input transitions for a state (*e.g.,*t0,t1,t2) and zero to many output transitions (*e.g.,*t2,t3,t4,t5). We will not focus on the input transitions because only the output transitions of cr_{st} are used in the construction of $[\![r_{op}^{fsm}]\!]$. Let T_o being the set of output transitions of the current state.

Note: in the following we do not detail the semantics of boolean expressions (which is a classical boolean algebra and integer comparison). We use the same facilities for expressions on integers.

State Machine

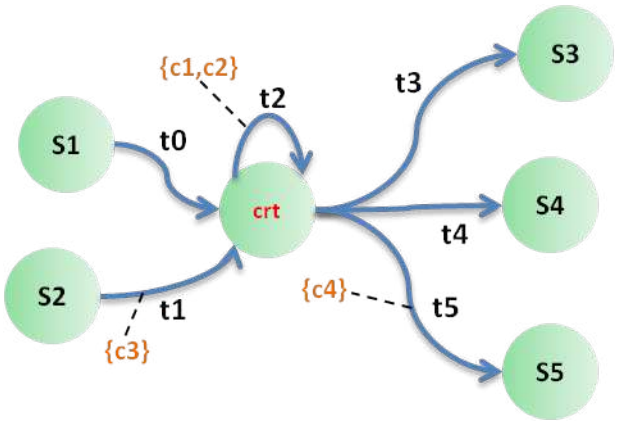


Figure 1: Partial automata Sample: crt = cr_{st}

$$[\![r_{op}^{fsm}]\!] = \vee_{t \in T_o} [\![t]\!] \quad \text{(state machine)} \tag{43}$$

The semantics of a r_{op}^{fsm} in a current state is defined as a *logical or* (logical disjunction) on the set T_o of output transitions for this specific current state taking into account the semantics associated to the transition t.

Transition

$$[\![t]\!] = [\![G]\!] \quad \text{(transition)} \tag{44}$$

$$\frac{\begin{array}{l} C_{True} \subset F \wedge \\ \forall c \in C_{False}, c \notin F \wedge \\ t.source = cr_{st} \end{array}}{cr_{st} = t.target \wedge [\![A]\!]} \quad \text{(transition rewriting)} \tag{45}$$

The semantics of a transition t is given by the coupling of: the semantics of the guard associated to it and the rule reifying the transition rewriting function between a source state and a target state. The transition rewriting function depends on a couple of premises. Considering $t.source$ as the cr_{st}, if $C_{True} \in F$ and $C_{False} \notin F$, then with the evalution of guard the next cr_{st} is $t.target$ and an action is potentially performed.

Guard

$$\frac{guard = [\![BE]\!]}{[\![< BE, C_{True}, C_{False} >]\!] = guard \wedge \left((\bigwedge_{c \in C_{True}} c) \wedge (\bigwedge_{c \in C_{False}} \neg c) \right)} \quad \text{(guard)} \tag{46}$$

For a given guard and its boolean expression. Depending on the value of the boolean expression, the $c \in C_{True}$ tick, when the $c \in C_{False}$ does not tick. When the $c \in C_{False}$ ticks or the guard evaluation is false, then the transition is not operated.

finish action

$$[\![kill(c)]\!] = (\mathcal{I}_c = \mathcal{I}_c + c^{\downarrow}) \quad \text{(finish action)} \tag{47}$$

A finish clock action ($kill(c)$) implies the death of the clock c.

3 Runtime State definition of a MoCCML state machine

A MoCCML state-based model has a state which evolves all along a schedule. This state relies on the definition of the history of the set of clocks. Naturally speaking, the "runtime state" of a state machine contains the current state of an instance of the state machine and the values of each of its local variables.

3.1 Configuration definition

The history of a MoCCML *Declarative Model* $\mathcal{M} = \langle \mathcal{C}, \mathcal{S} \rangle$ for a specific schedule σ has been defined as $H_\sigma : \mathcal{C} \times \mathbb{N} \to \mathbb{N}$ (see section 2.1.2).

The definition of a MoCCML *state-based model* history extends the previous definition with two more parameters, *i.e.,*: the current states of each state machines and the values of each local variable.

We expand the history H_σ into three combined functions and formalize it as follows:

$$H_\sigma \ ::= \begin{cases} H^c_\sigma \ : \mathcal{C} \times \mathbb{N} \to \mathbb{N} \text{ relation (1)} \\ H^q_\sigma \ : 2^Q \times \mathbb{N} \to 2^{cr_{st}} \text{ relation (2)} \\ H^{CE}_\sigma \ : 2^{CE} \times \mathbb{N} \to \mathbb{N} \text{ relation (3)} \end{cases} \tag{48}$$

The first relation is similar to the one defined in Section 2.1.2, it associates to a clock in $\mathcal{C}$ and a step number in σ the number of tick of the clock at this step. From now, the history H_σ from Section 2.1.2 is replaced with H^c_σ.

In the second relation, for a given set of set of state (*i.e.,*the states from all state machines $\in S$) and a step number in σ, H^q_σ provides the set of current state of each state machine.

The third relation says that for a given set of concrete entities (the union of the ones in the declaration blocks of each state machine $\in S$ and a given step number in σ), H^{CE}_σ provides the set of all the concrete entities values.

At each step in the history, the clocks $\in S$, the state machine $\in S$ and the concrete entities in the declaration blocks of the state machines $\in S$ are considered to determine the current history.

4 Towards an exhaustive exploration

The goal of this chapter is to define the approach which provides the capacity to explore the behavior of the MoCCML model associated with the DSML model. This functionality provides the exploration of all the possible behaviors of the MoCC associated with the DSML model.

During the Gemoc project, the simulation functionality is the main focus to obtain a trace of the model execution based on the language workbench tooling. Regarding the exploration, the simulation trace represents a subset of the exploration and so a subset of the behaviors of the MoCC. The exploration is possible in the scope of the mapping definition between the MoCC and the DSML. The definition of this mapping must take into account the capacity of the exploration and the constraints with it.

A MoCCML model exploration is necessarily based on a configuration definition and also a definition of an abstraction of the behavior of the DSE events and DSA actions from the explorer point of view. This abstraction definition must be defined to close the explored behavior.

The definitions of this chapter are the first attempt to define the methodology to explore the behaviors of the MoCCML model.

4.1 Configuration definition

A MoCCML state-based model configuration relies on the history of the set of clocks, the set of the current states of a *StateBasedRelationDefinition* and the values of the set of local variables.

In our case, these configurations are defined as the configurations of a Labeled Transition System where the states encode all the current states of the state machines and the current values of the RTD variables. The transitions between states are labeled by the events of the fired events in the state machine behaviors of the given *StateBasedRelationDefinition* .

In this context, the methodology consists of a selection of the events mapped to clocks and the selected variables that are defined as RTD. According to the definition of the section 3, the possible type of the variables is $Integer$.

The selection of the number of variables and clocks is very important to obtain an explorable statespace. So one of the methodology key features can be to provide guidelines to define amenable to exhaustive exploration.

4.2 DSML abstraction

The mapping between DSML and MoCCML is made at the ECL specification level. In this specification, we define events associated with the actions of the DSA and also events associated with the DSE events. On these event bindings we apply the MoCCML relations of the MoC Library to schedule the events. To make an exhaustive exploration of the finite state space of a system using such scheduling constraints, the idea is to abstract the DSA/DSE behaviors of the DSML in the form of state machines that will be synchronized with the MoCCML state-based relations instantiated on DSML models.

Figure 2 illustrates the elements that are taken into account to provide exhaustive exploration and simulation. The first two lines of the figure show the ECL mapping (⑤) definition which takes as inputs DSA (①), DSE (②) and the MoCCML relations (③). The ECL mapping is applied to the DSML models (④) to unfold all the relationships defined in the ECL mapping for this DSML instance. The resulting model is the DSML +MoCCML instance (⑦).

The state machines (⑥) corresponding to the behavior of the DSA/DSE are also generated from the ECL mapping (⑤). To stay within a purely state-based description style, these abstractions related to DSA/DSE behaviors are made in the form of StateBasedRelationDefinition.

The previous unfolding step, coupled to abstract state-based behaviors (DSA/DSE), allows building the finite state space that will be used for exhaustive exploration (eg OBP) or simulation (*e.g.,*TimeSquare).

5 Conclusion

This document presents the operational semantics of the MoCCML language which was defined to create MoCC models associated with any DSML.

This semantics is an extension of the CCSL language definition based on equational relations completed by state based relations.

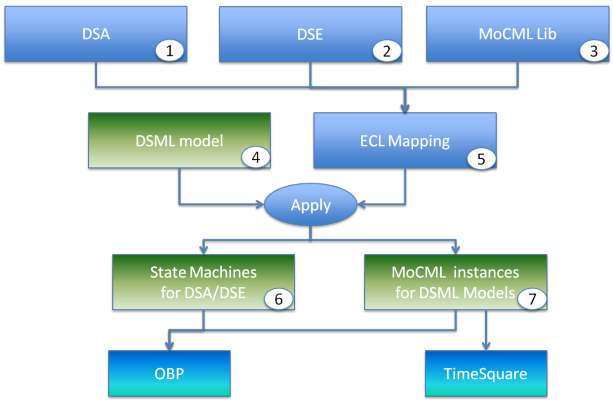


Figure 2: Mapping architecture overview

The formal semantics of the MoCCML language allows for a composition of the two kinds of relations and is amenable to simulation, *i.e.*,the construction of schedules , which satisfy the models. The semantics uses the notion of history, which is used as a basis to define the notion of "runtime state" itself under study to extend the tooling associated with MoCCML towards an exhaustive exploration of the MoCCML models.

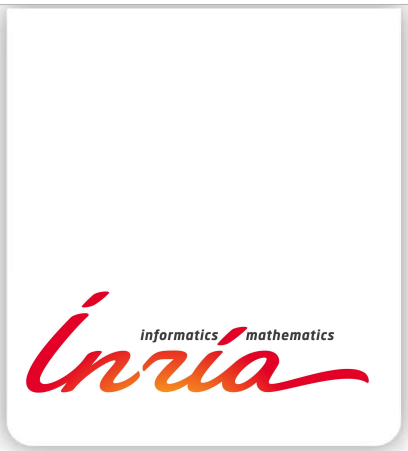
The tooling (MoCCML editor, TimeSquare, ECL language and the OBP explorer) around the MoCCML language must be aligned, or take into account, the definitions of this document.

References

References

[1] Charles André. Syntax and Semantics of the Clock Constraint Specification Language (CCSL). Research Report RR-6925, INRIA, 2009.

[2] GD Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61(January):17–139, 1981.



RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399