Grant ANR-12-INSE-0011

ANR INS GEMOC

D2.1.1 - Ecore-based metamodel of the MoCC modeling language

Task T2.1

Version 0.2

Authors

Author	Partner	Role
Julien DeAntoni	I3S / INRIA AOSTE	Lead author
Joel Champeau	Lab STICC / ENSTA Bretagne	Contributor
Jean Christophe Le Lann	Lab STICC / ENSTA Bretagne	Contributor
Papa Issa Diallo	Lab STICC / ENSTA Bretagne	Contributor

Contents

1. Introduction

1.1 Purpose

In the gemoc project, an executable Domain Specific Modeling Language (DSML) is represented as a tuple of modeling units *<Abstract Syntax, Domain Specific Action, Domain Specific Event , Model of Concurrency and Communication >* (see Figure 1.1) as detailed in the D.1.1 deliverable where AS is the abstract syntax, DSA are the Domain Specific Actions (defining both the execution state and the execution functions) and DSE are the Domain Specific Events that represent the mapping between the other units. In this context, a Model of Concurrency and Communication (MoCC) represents the concurrency, synchronizations and the possibly timed causalities in a language. It must represents the acceptable schedules of the atomic actions of the language, which represent both computation and communication.

This document presents the first steps towards a language (*i.e.,* an abstract syntax) dedicated to such a description. An important restriction for this language is to be independent of the Abstract Syntax of the language it schedules in order to allow its reuse with various language AS. It is based on two existing languages: `CCSL` , which is declarative and Cometa, which is operational. This document is the first version of the fusion of two languages in order to get benefits from both of them. It will be update after the first experiments.

1.2 Perimeter

The document includes the presentation of CCSL metamodel, the Cometa metamodel, the integration strategy of the both metamodel and the currently identified MoCC language use cases. This document is the first attempt of the integration of the two metamodels.

1.3 Definitions, Acronyms and Abbreviations

- MoCC: Model of Concurrency and Communication (even if the meaning of MoCC is more associated to Model of Computation and Communication)
- CCSL: Clock Constraint Specification Language
- AS: Abstract Syntax
- DSA: Domain Specific Actions
- DSE: Domain Specific Event
- Cometa: Communication Metamodel

1.4 Summary

The main purpose of this document is to provide an overview of the MoCC metamodel based on CCSL and Cometa metamodels. This document is in line with the MoCC Ecore file. The organisation of this document includes in the first section a presentation of the CCSL metamodel, following by a presentation of the cometa metamodel. The metamodel integration strategy is presented after and the next section defines the current identified use cases to use this MoCC

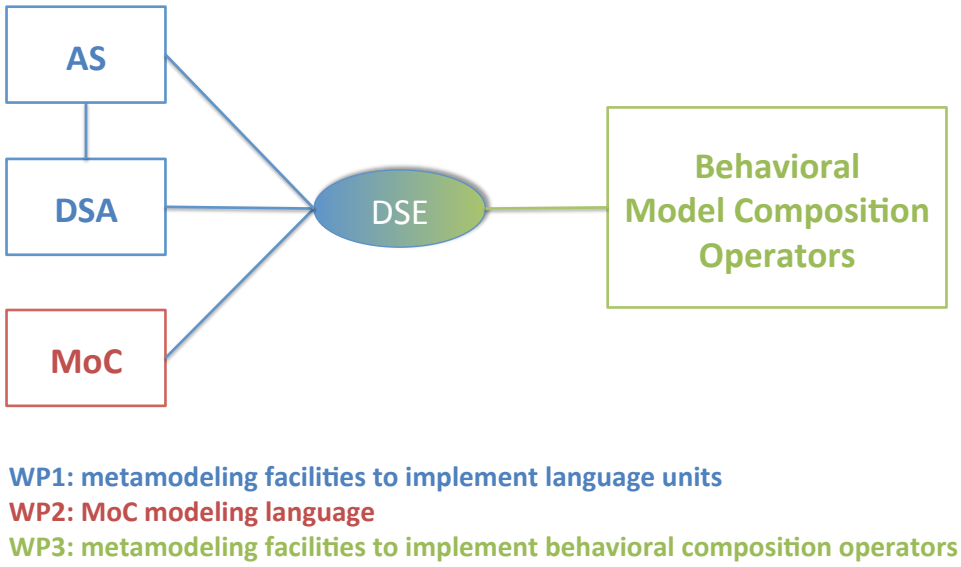

Figure 1.1: the Modeling units in GeMoC

language and the document ends by a conclusion.

2. Model of... Computation ?

The state of the art gives plethora of definition of what a MoC is or isn't. There is no real consensus on the definition so we first give a view of our understanding of what a MoC is. [4] has given a definition of a Model of Computation (and Communication): "*How the abstract machine in an operational semantics can behave is a feature of what we call the Model of Computation of the language. The kind of relations that are possible in a denotational semantics is also a feature of the model of computation.*". The authors discuss features of the Model of Computation including the communication style, how individual homogeneous behaviors are aggregated to make complex compositions and how concurrency is handled. A different definition is given by [5], which define the model of computation provided by a modeling language as: (1) the model of time employed (untimed, integer-valued, etc) and the event ordering constraints within the system (globally ordered, partially ordered, etc), (2) the supported method(s) of communication between concurrent processes and (3) the rules for process activation. From these two definitions, a proposed simple definition is "a MoC is defined in terms of language elements, the concurrency, synchronizations and the, possibly timed, way they interact together during the execution of the system (causalities)".

Because we split a language into units where the AS and DSA are specified independently of the MoC, we want to specify the MoC indepndtly of the other units. It makes the MoC closer to the one idea developed in concurrency theory from which MoC has been inspired. In concurrency theory the focus is made on the event structure that represent the timed and concurrent schedule of the system [9, 6, 7]. To conclude, in geMoC, the MoC is really a model of concurrency without any computational concern. More precisely it is a specification of the possible, possibly timed and concurrent, causalities in a system.

2.1 CCSL

2.1.1 Rationale and overview of the approach

In CCSL, interactions between events are specified using logical and multiform time. Logical and multiform time allows considering not only the distance between two occurrences of events (that would be expressed relative to the physical

time) but also the relative ordering of event occurrences. An interaction is then seen as a set of partially ordered set of events that defines a set of acceptable behaviors. Then, if required, one possible behavior may be selected according to different criteria on non-functional properties (like reducing the memory footprint). For instance, considering a producer / consumer system, the producer and consumer activations are modeled as two events on which a logical relation imposes that an item has to be produced before being consumed. If we then consider reducing the size of the communication queue, future activations of the producer must then be delayed until the consumer has consumed all available items.

In CCSL (Clock Constraint Specification Language) , a Clock is a possibly infinite ordered set of instants. Each instant represents an event occurrence. Based on this, CCSL is a model-based declarative language, which can be used to specify multiform and logical relations between the possibly infinite clocks (*i.e.,* event) CCSL has a formal operational semantics [1] that defines valid executions according to a given CCSL specification or reject the specification when there are some contradictions. The computation of valid executions is called *clock calculus*.

A CCSL specification is a set of clocks and clock constraints that formally defines a set of possible interactions. Foundational constraints are defined in a kernel library. CCSL allows the definition of user-defined constraints by combining existing relations imported from a set of kernel relations or from another user-defined library. These user-defined relations can be grouped in a library. All the computation and communication rules for a given MoCC are grouped together in a MoCC library. A MoCC library is a reusable entity that can be used by several CCSL specifications.

A MoCC is the definition of constraints on events that should be made explicit in the specification of a language. When we have a model that conform a language, then an execution model is the set of constraints and events actually used according to the model structure. In this way an execution model is a CCSL specification that uses one or more MoCC libraries. An execution model is usually suitable to simulation, for our case in the CCSL associated framework, named TIMESQUARE . The result is directly linked to the specification and provides interesting interactive feedback such as the drawing of timing or sequence diagrams, or the animation of existing diagrams. TIMESQUARE also provides a mechanism to define new ways to interpret the results of the clock calculus, *i.e.,* to build user-defined back-ends [3].

After this overview of the proposed approach, the following sections elaborate on different aspects of CCSL . The explanation starts with a informal description of CCSL and its library mechanism.

2.1.2 CCSL *abstract syntax*

In this paper, we present only a small subset of CCSL , focusing on syntactic and semantic aspects sufficient to understand the idea developed in the AOSTE team. The syntax and the semantics of the whole CCSL language is available in a technical report [1].

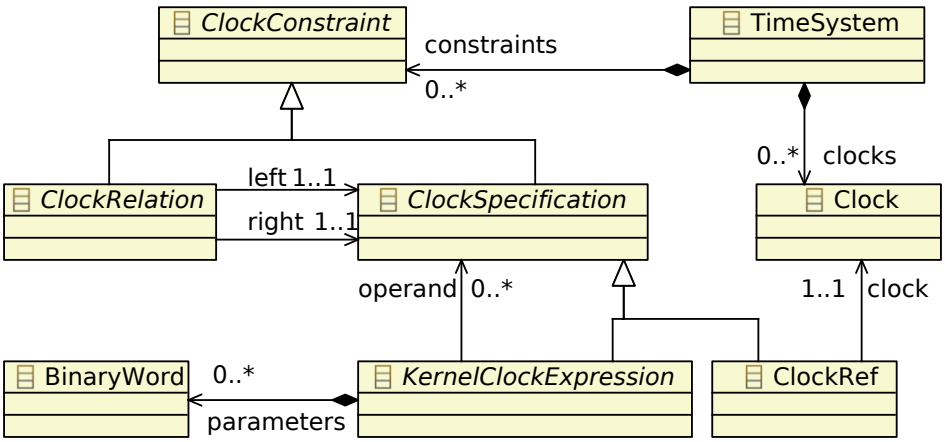

Figure 2.1: Simplified classical CCSL *metamodel*

A first view of the CCSL abstract syntax is represented in Figure 2.1. A concrete syntax is also given to make the illustration easier to understand. A *TimeSystem* (aka CCSL specification) consists of a finite set of *Clocks*, and the parallel composition of a set of *ClockConstraints* (denoted |). A clock is a strictly ordered set of instants, generally infinite. Two instants belonging to different clocks are possibly related by a precedence ($\prec$) or a coincidence ($\equiv$) relationship. As a result, a time system can be seen as a partially ordered set of instants [2]. A clock constraint specifies generic

associations between (infinitely) many instants of the constrained clocks. A clock constraint is either a *ClockRelation* or a *ClockSpecification*. A clock relation mutually constrains a right and a left clock specification. A clock specification is either a *ClockExpression* or a simple reference to a clock (*ClockRef*. A clock expression may have parameters and specifies a new clock according to the kind of expressions and its parameters. Parameters here are simply represented by a, possibly infinite, *BinaryWord*.

In this paper, we consider only two kinds of clock relations:

- the precedence (denoted $\prec$), which specifies that $\forall k \in \mathbb{N}$ the k^{th} instant of the left clock appears before the k^{th} instant of the right clock.
- the coincidence (denoted $=$), which specifies that $\forall k \in \mathbb{N}$ the k^{th} instant of the left clock is simultaneous to the k^{th} instant of the right clock.

The only clock expression of the example is the *filteredBy* expression (denoted $\blacktriangledown$), which takes a clock reference as an operand and a binary word as a parameter. It specifies that the resulting clock is the clock operand pruned of some instants. Which instant is kept and which is pruned depends on the binary word parameter.

Example We assume two clocks A,B constrained by the CCSL specification $\mathcal{S}$ below.

$$\mathcal{S} = \left(A \prec B \blacktriangledown (10)^{\omega}\right)$$

$(01)^{\omega}$ denotes the infinite (periodic) binary word $0101\cdots01\cdots$. This specification is slightly reformulated to facilitate the semantic explanations: we introduced one implicit clock C that evolves in coincidence with the *filterBy* clock expressions of the previous specification.

$$\mathcal{S} = C = B \blacktriangledown (10)^{\omega} \mid A \prec C$$

2.1.3 CCSL *semantics*

This paragraph gives the flavor of the CCSL semantics without falling into the details. If the reader wants the whole formal operational semantics, he can refers to [1] where the semantics is given mathematically in terms of Structural Operational Semantics rules [8]. A CCSL specification is an executable model. An execution of a CCSL specification is an infinite sequence of reaction steps. Each reaction step computes the set of clock that can/must/cannot "ticks" (*i.e.,* the set of events that can/must/cannot occurs) to represent the evolution of the system. This set is computed according to the specified clock constraints. The computation is based on the resolution of a boolean expression defined by the conjunction of the boolean expressions induced by the clock constraints of the system. Consequently, each clock constraint corresponds to a boolean expression. Once a reaction is computed, the system evolves by propagating the result of the reaction in the system.

Pragmatically speaking, the semantics of CCSL is based on two main operations any constraint should implement:

- `Constraint::getSemantics(): BDD`
- `Constraint::updateInternalState(ClocksThatAreTicking: oSet<Clock>)`

The first one asks all clock constraints to compute the boolean expression implied by their semantics and current internal state. It returns a mathematical representation of the clock constraints as a Binary Decision Diagram (BDD). The second one ask the definition to update its internal state according to the set of clock that actually ticks.

2.1.4 Introducing user-defined constraints in the CCSL *abstract syntax*

Using a metamodel like the one presented on Figure 2.1 (of course augmented to represent the whole CCSL kernel) allows user-defined CCSL specifications. However, it is not possible to create reusable composition of clock constraints to specify a MoCC library. What is expected is to define the equivalent of reusable parametrized modules, which can be defined and then instantiated, when necessary, with the appropriate parameters. For that purpose, we propose the metamodel presented in Figure 2.2, supporting the construction of user-defined CCSL libraries (instantiated from *CCSL_Library*). This metamodel is a simplified version of the one actually used, and yet, it is sufficient to understand the principle. A library is a set of *Definition*s. A definition specifies a reusable module. A definition contains entities that

can be either concrete or abstract. An *AbstractEntity* is a parameter of the module. A *ConcreteEntity* can be considered as an existing object in a classical objec t-oriented programming language. A *ConcreteClockRelation* is a concrete entity defined by a *ClockRelationDefinition*. It is a specific use of the referenced clock relation definition. It can be used in two contexts: either in a definition to add some definition internal relations or in a CCSL specification to specify the use of a specific relation. When used in a CCSL specification, a concrete relation binds the *Variable*s of its definition with some concrete elements.

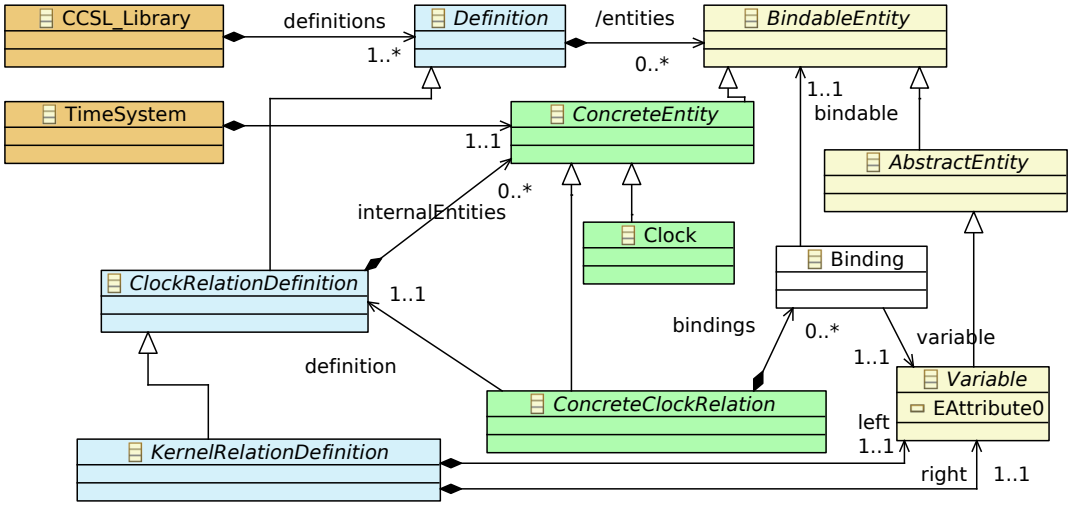

Figure 2.2: Simplified extensible CCSL *metamodel*

In this case, a *Binding* links together a variable and a concrete entity. When a concrete relation is used in a definition, a binding can link the variables of their containing concrete relations either to a concrete entity or to variables of the definition where it is used. For this reason, a binding links together a variable with either another variable or a concrete entity depending on the case.

An important point is that we want to provide an homogeneous way to define CCSL specifications (instances of TimeSystem) either based on CCSL predefined constraints or on user-defined libraries. Consequently, all the predefined constraints have been described in the metamodel as specific definitions and their instances have been grouped together in a kernel library. With this metamodel, a CCSL specification is made up with concrete entities equally defined by definitions from the kernel library or from user-defined libraries. In this case, the two operations on constraints are now defined on the Definition concept. This is really important because it means we have an homogeneous representation of an execution model independently of the fact it is an execution model importing a single MoCC library or several ones. Also a direct benefit is the fact that the definition semantics is on boolean equations and we could imagine to change the internal language of definition as long as the language is (formally) able to give the boolean equation that results from the parameter and its internal state.

Figure 2.3 illustrates the usage of this metamodel. It represents a CCSL specification that contains a concrete clock relation (*ccr1*) defined as a user-defined relation. The user-defined relation (named $\boxed{3=}$) specifies a ternary coincidence relation and consequently possesses three variables (*v1*, *v2* and *v3*). This definition contains two internal concrete relations (*rel1* and *rel2*) whose definitions (named $\boxed{=}$) are given in the kernel library. In the CCSL specification, the user-defined relation is used with three clocks as parameters: *c1*, *c2* and *c3*. The textual specification of the $\boxed{3=}$ relation is the following:

$$\begin{aligned} &\text{def } \boxed{3=} \text{ (clock } v1, \text{ clock } v2, \text{ clock } v3) \triangleq \\ &\quad v1 \ \boxed{=}\ v2 \\ &\quad v1 \ \boxed{=}\ v3 \end{aligned}$$

To ease the presentation, the metamodel presents only clock relations. Exactly the same pattern composed of definitions, concrete entities, variables and binding is applied for clock expressions. In the proposed approach, the user-

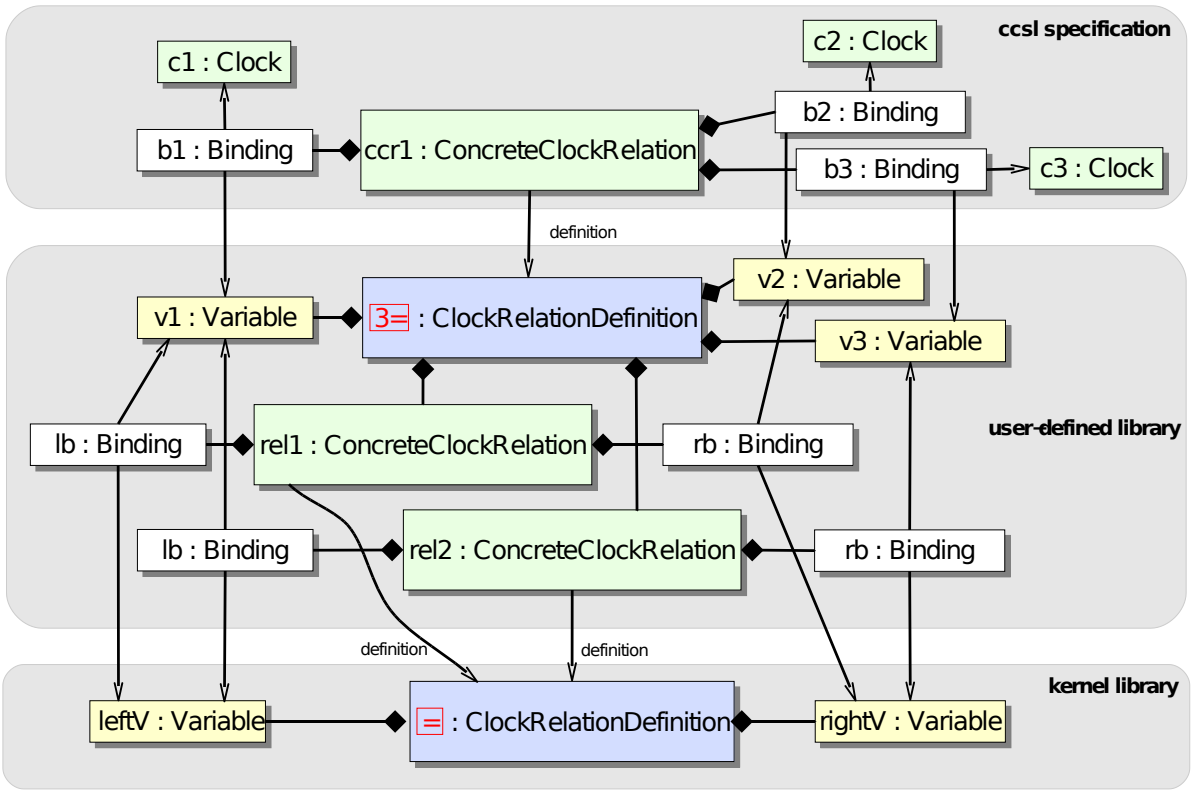

Figure 2.3: An example of user-defined library (in the middle) based on the kernel library (in the bottom), and its use (in the top)

defined libraries are used to specify MoCC interaction libraries. These libraries must be executable and must conform to the operational semantics.

3. Cometa metamodel

The Cometa DSML allows describing, operationally, the control of the interactions between concurrent entities with respect to the communication strategy inferred by a given MoCC. The MoCC strategies are specified within separate libraries of MoCC-based behaviors. The behaviors are responsible of the definition of the operational semantics of the selected MoCC.

3.1 Overview

In Cometa, the operational semantics is defined by communicating state machines. This approach is due to the need to specify the semantics of the concurrency and the communication in a language which include parallel entities. One the first goal of the Cometa approach is to generate code for parallel entities of languages and targets parallel and heterogeneous platforms. In this context, Cometa was applied on languages integrating the component concept where each concurrent entity is able to exchange (*write* / *read*) data with a given other concurrent entity. For each communication request based on *write* / *read*, the Cometa model assumes that the communication, through the connector between the 2 components, respects the selected MoCC like KPN or CSP, for example. The *write* / *read* are the relevant concepts in the dedicated DSML to interface the MoCC and so with the Cometa model.

Each concept *write* / *read* is mapped to MoCC event and triggers the MoCC events of the Cometa model which is a composition of several state machines which are the definition of the MoCC. The MoCC events and state machines control the execution of the concurrent entities of the DSML as presented in the figure 3.1. In the Cometa approach the execution control is based on a network of the state machines to ease the code generation of this control on several platform and particularly for parallel heterogeneous platform.

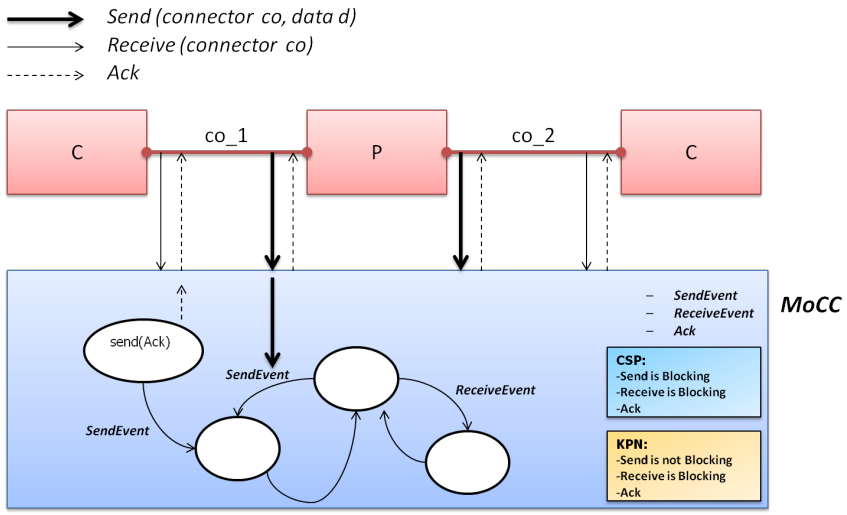

Figure 3.1: Examples of MoC behavior definitions with Cometa.

In Figure 3.1, the operational semantics of the MoC specify that *write* / *read* requests the behavior of the Cometa model based on MoCC events, state machines (for illustration purpose, the set of state machine is abstract by only one state machine) and for some MoCC definition some dedicated structure like FIFO. All the possible events and structures are presented in the next sections.

In the first example, the implemented FSM follows the KPN MoCC rules. The underlying operational semantics of the MoCC suggests that send requests are non-blocking and receive requests are blocking if the memory is empty (empty

FIFO). From this property, we deduce that the implementation of the MoCC FSM must guarantee an instantaneous "Ack" response to each producer P that enables a SendEvent. From the reception of a ReceiveEvent, an "Ack" will be sent to retrieve the requested data if the data is available in the FIFO (a checkFIFO function returns the information).

In the second example, the MoCC implemented is CSP. The underlying operational semantics of the MoCC suggests that send and receive requests are synchronized, therefore they are blocking. There is no need here of a FIFO for storing data. Usually, shared variables are sufficient for the communication. In this context, the implementation of the MoCC FSM must ensure that *Ack* are sent to the reader and the writer.

To express a set of state mchaines with Cometa, we support several aspects in the Cometa metamodel. The structure part to define the organization of the state machines, the behavior part to define each state machine and the library aspect to create reusable MoC definitions. In the scope of this document, we focus on the structure part and after the behavior of the metamodel in the next sections.

3.2 The structure aspect

The specification of a structure highlights the level of parallelism in an interconnection of communicating components. By level of parallelism we mean the identification of the parts in the system structure that remain parallel once the execution control behaviors have been applied in the model. Indeed, the execution control mechanism allows introducing "sequential" execution between some parts (entities) of the system. This structure description also eases the mapping of control machines by clearly separating the control behaviors that are for protocols and those for scheduling. The abstracted concepts are presented in the figure 3.2.

3.2.1 The Structure element

A *StructureElement* is an abstract class that defines the common properties of the elements for the description of topology (MoCComponent, MoCPort, and MoCConnector). The common properties are among others the references to control behaviors and the ability to define variables (parameters). The structure aspect of the metamodel is presented in the figure 3.2

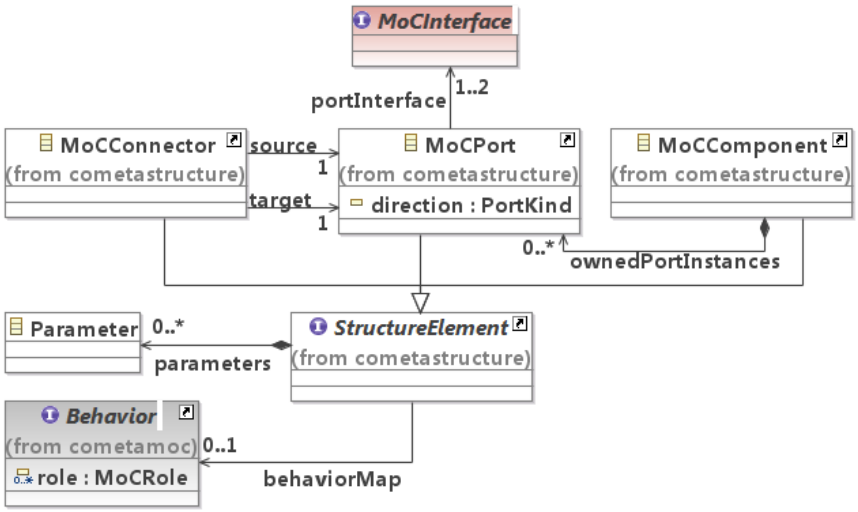

Figure 3.2: Structure aspect of the metamodel

3.2.2 The component concept

A MoCComponent is a virtual component that can carry the control behavior of application components. The behaviors defined for this type of component are *scheduling* ones; what differentiates them from behaviors defined in the communication protocols. They are linked to application components through MoCPort. The MoCComponent allow setting

up the structure accommodating the functional blocks of the application model, and they provide support for a hierarchical description of components thanks to the use of CompositeComponent. They are of two types: BasicComponent, CompositeComponent.

- BasicComponent is a concurrent unit which has no internal hierarchy. It is the MoCComponent corresponding to an atomic concurrent element. The functional blocks of computation (potentially described by another language) can be allocated on BasicComponent. They have ports for communication. In this approach, we assume that the function blocks are executed sequentially.

- A CompositeComponent allows introducing the notion of hierarchy of components. It is a concurrent entity that contains other CompositeComponent, or BasicComponent. It also contains ports and connectors for communication between the various components it contains. As for BasicComponent, MoCC based behavior can be applied to composite components for general scheduling purposes.

3.2.3 The port concept

A MoCPort is used for the communication between components. It is either the entry or exit point for data from one component to another. The MoCPort can be enriched with MoCC based behaviors. When applying a MoCC behavior to a MoCPort, this behavior participates in the specification of the communication protocol.

The Events (generated) from the application requests are sent to the execution control behaviors via MoCPort. They define interfaces (MoCInterface) that are contracts to guarantee the access to the internal MoCPort control behavior (protocol) or to guarantee the access to the behavior of the component to which the MoCPort is attached (MoCConnector / MoCComponent). A MoCPort can be directly connected to another access point (MoCPort) of another component (see section 4.5). Here, the control behavior is divided into two interconnected machines. The MoCPort can be oriented in / out / inout to specify the direction of the communication. The Behavior and interfaces of MoCPort constitute its specification.

3.2.4 The connector concept

A MoCConnector is a virtual component that allows connecting two communicating concurrent components. Here, the execution control layer (between two concurrent entities) defines a behavior that can be divided and mapped respectively into two MoCPort (in / out) and the MoCConnector. If such a control behavior is represented by a single FSM, then the machine is carried by the MoCConnector. The MoCConnector behavior is related to the machines of the two MoCPort (in/out) via references of type (source / target). The MoCConnector can be related to memories or to buffers to store the data.

3.3 The behavior aspect

In this metamodel aspect, we define the state machines which are the support of the specification of the protocol between the MoCComponents.

3.3.1 The state machine

Cometa defines event-based FSMs. The event-based FSMs are machines for which the variables or symbols that allow the changes of state are events from the system. As stated earlier, in the context of communicating concurrent entities, such events are requests for send / or receive of data.

Cometa state machines allow defining a set of control states. The machines allow defining (as restriction) a processing order of the requests that are sent to the machine. The state changes represent the different steps of control underlying a type of communication formally defined by MoCC rules.

With the abstracted concepts it is possible to describe both types of machines (Moore, Mealey). The choice of one or the other machines to capture a control behavior is left to the discretion of the user based on the constraints he faces in terms of implementation. We will see the description of the FSM concepts in the following sections (see Figure 3.3.

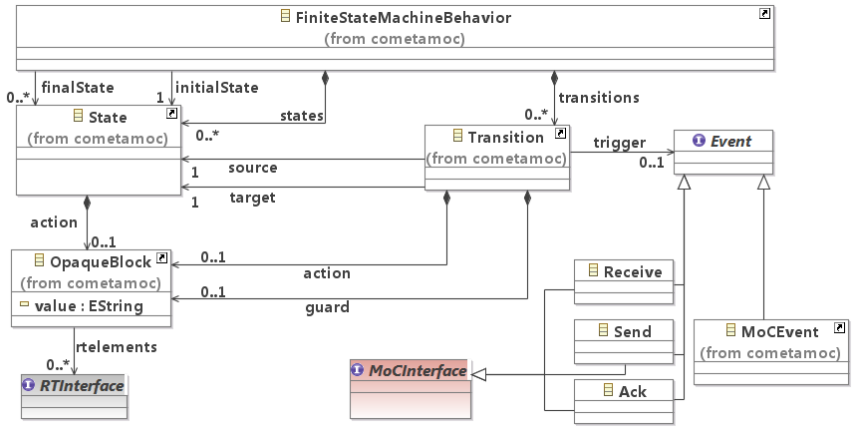

Figure 3.3: The behavior aspect of Cometa

3.3.2 The state concept

The notion of *State* in Cometa is a description of a state of control. A control state contributes in regulating incoming and outgoing requests of various concurrent entities. It allows defining the order in which the various requests must precede or succeed. This is done by describing in each state the authorized events / requests, and by defining the successors and predecessors states (of each state) that are part of the synchronization process.

On this basis, it is possible to define the "communication patterns" allowed between the entities (i.e. their MoCC). The set of states of a control machine, and their transitions define the interaction patterns. An interaction pattern shows a possible way of communication and exchange between the entities of the system.

In Cometa FSM, the changing of states can imply a sequence of instructions to be executed either on the target state or on the transition using the notion of a *Block*.

3.3.3 The transition concept

The concept of "Transition" is used to describe the relationships between the various states of the control FSM. They are used to define relations of succession and precedence between the control states. Specifically, Cometa transitions define references source, target to denote the "source" states and "target" states for a given transition.

Thus, starting from the set of transitions between control states, one can deduce all relations of type predecessor, successor between states. Transitions have "trigger" reference that is of type "Event". The occurrence of an event called the "trigger" may involve state change during the control process if all additional conditions on the transition are verified.

The conditions on transitions are referenced as the "guard". A "guard" is placed on the transition expression and its evaluation is mandatory for the state change.

- Trigger: This reference is used to determine the type of event occurrences that may cause a change of state. In the context of Cometa, occurrences of events that are "trigger" are MoCInterface (Send / Receive / Ack) and MoCEvent (see MoCInterface section). The objective is to provide an order of processing requests by the synchronization rules.

- Guard: The "guard" are references to expressions to evaluate. The expressions are additional conditions of transition from one state to another. In the context of the Cometa "guard" references "Block". The return of the evaluation of the Block content potentially authorizes the change of state.

- Action: The "action" reference is defined for the "State" and "Transition". In both cases, they are references to "Block". The Blocks define the sequences of instructions for internal and external communication. The internal communication is the communication between control FSM and external control is the answer provided to each requesting concurrent entity. We will see the description of the contents of "Block" in the RTInterface section.

3.3.4 The Block concept

A *Block* defines a sequence of instructions. The defined instructions allow communication between the control FSMs and the access to shared memory through the primitives (services) defined in "RTInterface." A "Block" has a body attribute that is a string representing the sequence of instructions. In a sequence of instructions one can call services to gain access the shared memory (add, remove on FIFO for example); or one can call functions that convey MoCEvent (for internal communication) and MoCInterface (for external communication).

3.4 The specification of the Cometa interfaces

The interfaces that we address in this section have a dual purpose. They serve as a communication contract between the layer of MoCC-FSM and the concurrent entities; they also serve as access to the run time engine which can be virtual or connected to a real implantation. We define two types of interfaces, MoCInterface to meet the first objective and RTInterface to meet the latter objective. The metamodel aspect of the definition of the interfaces is prented on the figure 3.4

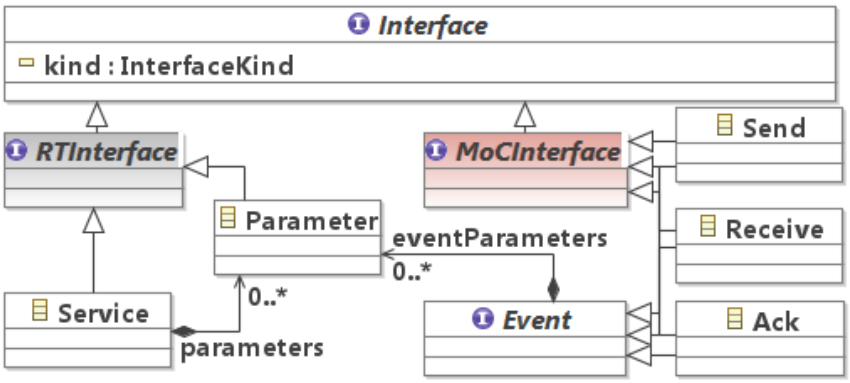

Figure 3.4: The behavior aspect of Cometa

3.4.1 The specification of the MoC Interface

The notion of MoCInterface captures the subset of events that represent produced requests between the MoCC layer and the concurrent entities. We have restricted this subset to three types of exchanges (interfaces): Send and Receive requests (from entities models to the MoCC models), and Ack (from the MoCC models to entities model). On one hand, the occurrence of events SendEvent, ReceiveEvent are used to trigger state changes in the control mechanism. On the other hand, the Ack release the components to continue their computation.

3.4.1.1 The send event concept

This abstraction allows describing the sending events. The sending events are the result of data sending requests emitted by an entity behavior. These events can be parametrized. The parameters are used to specify the receiver of the request, the communication connector, or the data to be transmitted, etc. Within Cometa, the events are used as "trigger" on the transitions of the control machines.

3.4.1.2 The receive event concept

This abstraction allows describing the reception events. The reception events are the result of the reception requests issued by an entity behavior. These events can be parametrized. The parameters are used to specify the communication connector. Within Cometa, the events are used as "trigger" on the transitions of the control machines.

3.4.1.3 The ack event concept

The Ack events are sent back to the behavioral entity so they can continue their execution. The Ack are needed in all running control scenarios with Cometa. They reify the notion of permission to run. The FSM captures the control behavior and decides the sending of Ack according to the MoCC rules. For example, depending on MoCC rules, the sending of Ack can be unconditional on reception of a SendEvent or ReceiveEvent; elsewhere it can be done with some delay when a request must be blocked. These events can be parameterized. The parameters are used to specify the receiver of the request, the communication connector, or the data to be transmitted, etc.

3.4.2 The specification of the RT Interface

The notion of RTInterface captures the events (MoCEvent), as well as services and parameters that are used within the control behavior to interface with the communication media (shared memory such as FIFO and LIFO). Elsewhere, the services are used in the same way to store new events in the event queues that are operated by the schedulers.

The capture of services does not correspond with their implementation. Their implementation is provided as a library of functions external to the Cometa DSML. The captured services are used in the *Block* of the control behaviors as a mean to call their real implementation in the runtime.

3.4.2.1 The service concept

The concept of service is used in the context of Cometa to abstract functions interfacing with the "runtime", but also to abstract functions for communication between machines of control. In our experiments, we have identified three types of primitives for the above intentions. The primitive "sendOfEvent" is used for sending MoCEvent between control machines; the primitives add, remove, check are used to operate on an event/data queue.

3.4.2.2 The queue concept

The analysis of communicating behavioral entities must take into account the sizes of the used shared memories because in reality such sizes cannot be infinite. The notion of Queue is an abstract concept for the virtual representation of the shared memories such as FIFO, LIFO. The purpose of this virtualization is to highlight their sizes and the required services to operate on the Queues.

4. Integration Strategy

From our experiments, we tried to identify the best part of each languages. Quickly speaking, an execution in CCSL is declarative and consequently the refinement is simple (each constraint added refine the previous model). However the declarative style of CCSL makes (very) complex the description of some simple protocols.

The approach to create the MoC language metamodel consists in integrating the possibility to specify the semantics of some CCSL definition by using the Cometa state machines. It will keep the benefits of a declarative language for the specification of an execution model while providing an operational mechanism to specify constraint definitions in a MoCC library. Also, it makes the resulting language able to specify a MoCC library and an execution model independently of any targeted DSML abstract syntax. A simplified integrated model is presented on picture 4.1 The first step of our strategy is to create a MoCC language metamodel to aggregate all the necessary concepts and to complete this metamodel with the lacking concepts identified during experiments.

The approach is supported by the next two observations:

- Cometa state-machine manipulates states and variables. Since constraints can already be expressed in CCSL , the integration extends CCSL by Cometa state-machine and a some selected statements which are in discussion for now. It appends statements and control structures to manipulate the internal state of some definitions.

- the CCSL language is based on clock and clock constraints. A CCSL clocks is a possibly infinite ordered set of instants. The constraints define the relative partial order of the clock instants. So the MoC events of the Cometa

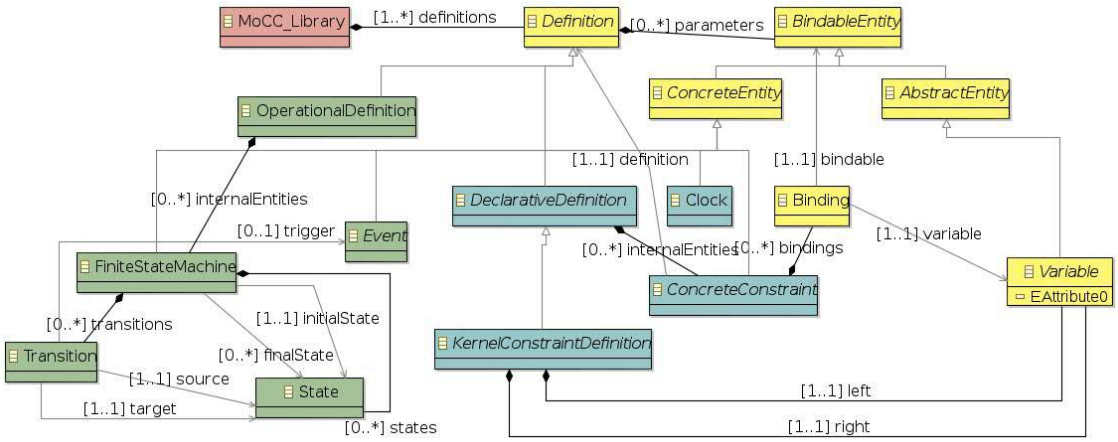

Figure 4.1: a simplified version of the MoCC language, integrating declarative and operational definitions

state machines follow the same definition than the CCSL clocks considering that there is an order preserving bijection between the cometa event occurrences and the instant of a CCSL clock.

This integration is a first version of the MoC language which will be refined with additional concepts after the first experiments. More precisely, during the project, we will consolidate the metamodel after each experiment to obtain the final version.

5. MoCC definition use cases

Cometa is a metamodel allowing the description of systems in terms of block diagrams, associated with finite state machines attached on both ports and channels. These FSMs are supposed to express Models of computation. Two remarks are needed : first, the MoCs are considered as channel caracteristics, and not global behavior per se. If a global behavior need to be described, Cometa user will be forced to think accordingly and make choices in terms of MoCs on various channels. This hard task calls for supplemental tools easing the fine tuning of these MoCs. This tool developped in the scope of GEMOC project is named NEMO. Secondly, this style if fundamentally operational: FSMs describe the mecanisms that produce events and the mecanims the consume these events. In the history of Cometa team, these FSMs describes the operational mechanism that simplify the implementation of the protocol state machines.

On the other hand, WP2 proposes to provide user with supplemental means to express MoCs using the more declarative approach of CCSL, perfectly adequate for specification purposes. Here CCSL is envisioned as equations giving precise synchronisations between infinite streams of data : inclusion of events, exclusions, alternance, etc. CCSL users have expressed the need to facilitate the expression of FSMs (e.g for protocol-centric applications), that is lacking in CCSL.

The unification of these two approaches is the main objective of GEMOC WP2. While the juxtaposition of the two metamodels is trivial in terms of technologies (both are developped using Ecore/Java technologies), it raises several questions concerning the supplemental descriptive capacities and their real compatibilities. For instance, due to their respective natures (asynchronous for Cometa and causal and polychronous for CCSL), the automatic exploration of mixed COMETA/CCSL using NEMO or TIMESQUARE is, today, impossible and the semantics translation is closed to the one described in [10].

In order to pave the way for a clean and convincing unification, we have selected 4 different use cases of MoC Langage description developped in WP2. These 4 use cases are depicted on figure 5.1, and described in the next sentences:

1. The first-use case makes the hypothesis that the MoC definition will solely rely on CCSL: then a MoC is simply

a set of constraints on clocks. Cometa is used as backend to run a generated automata corresponding to these equations. This automata is expressed in COMETA metamodel.

2. The second use-case instead starts with a purely FSM-based description of the MoC. CCSL is used as assertions to explicit some properties to be checked (during execution, model checking etc).

3. The third use case use FSM-based descriptions of the MoCC at the boundaries of components. In this case, the components are composed using CCSL equations, that will relate clocks of the outputs to the clocks of inputs. This set of equations can be transformed into a third FSM that acts as scheduler.

4. A final fourth case also depicted, named "mode automata" : here, each state is associated with a set of CCSL equations, active only when the state is the current state.

Also, for simulation purpose, it is planned to add the sematnics of state machine in the TIMESQUARE tool. This can be made by implementing the two methods associated with the TIMESQUARE solver for the `OperationalDefinition` concept. It will allows the conjoint and transparent use of mix description in one or several MoCC library.

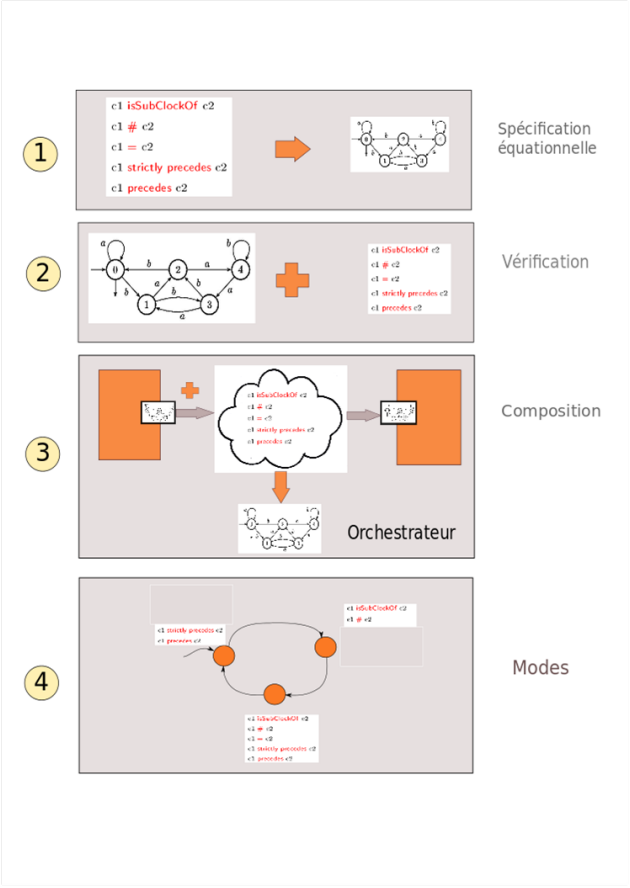

Figure 5.1: Possible MoC language use cases

We expect to exemplify these four use-cases in the remaining of the project related to the industrial DSML languages

defining in the WP5.

6. Conclusion and Perspectives

This first delivrable has allowed us the propose some structuring definitions, required to integrate both COMETA and CCSL approaches. The final goal is to provide a MoCC language with an apropriate usability: in many situation, state machines (associated with COMETA) or pure CCSL constraints are not sufficient nor practical. The fusion of the two descriptive styles is supposed to provide the user with specification facilities. We envision that the next steps in WP2 will be to define a concrete syntax for WP2 language, but more importantly to experiment WP2 language in various situations. These early experiments are needed to ensure that the expected usability is indeed effective. The supplemental capabilities of COMETA will then be used to explore some slight semantic variations while TIMESQUARE will be able to define MoCC library based both on declarative or operational definitions.

7. References

[1] C. André. Syntax and Semantics of the Clock Constraint Specification Language (CCSL). Research report, INRIA and University of Nice, May 2009.

[2] C. André, F. Mallet, and R. de Simone. Modeling time(s). In G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, editors, *MoDELS*, volume 4735 of *Lecture Notes in Computer Science*, pages 559–573. Springer, 2007.

[3] Julien DeAntoni and Frédéric Mallet. Timesquare: treat your models with logical time. In *Objects, Models, Components, Patterns*, pages 34–41. Springer, 2012.

[4] S. Edwards, L. Lavagno, EA Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: formal models, validation, and synthesis. *Proc. of the IEEE*, 85(3):366–390, 1997.

[5] T. Grötker. *System design with SystemC*. Kluwer Academic Publishers, 2002.

[6] Edward A Lee and Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(12):1217–1229, 1998.

[7] Antoni Mazurkiewicz. Trace theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, pages 278–324. Springer, 1987.

[8] G. D. Plotkin. A structural approach to operational semantics, 1981.

[9] Glynn Winskel. *Event structure semantics for CCS and related languages*. Springer, 1982.

[10] Ling Yin, F. Mallet, and Jing Liu. Verification of marte/ccsl time requirements in promela/spin. In *Engineering of Complex Computer Systems (ICECCS), 2011 16th IEEE International Conference on*, pages 65–74, 2011.

