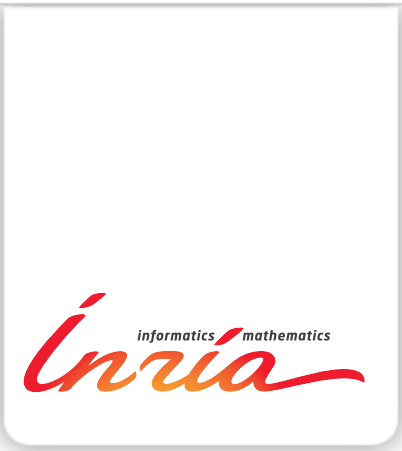


ECL: the Event Constraint Language

Julien DeAntoni, Frédéric Mallet

RESEARCH REPORT
N° 8031
July 2012
Project-Team Aoste

ISSN 0249-6399 ISRN INRIA/RR--8031--FR+ENG



ECL: the Event Constraint Language

Julien DeAntoni, Frédéric Mallet*

Project-Team Aoste

Research Report n° 8031 — July 2012 — 21 pages

Abstract: In Domain Specific Modeling Languages, the metamodel captures the domain concepts and their relationships. The Object Constraint Language (OCL) can be used to add structural invariants or to describe pre and post conditions on method calls. However, both OCL and current metamodeling languages fail in specifying concurrency, causality relationships and timed behavior of the models.
In this paper, we present a model-driven approach to a formal and explicit specification of causal and timed relationships within models by extending the OCL. We describe our domain-specific modeling language dedicated to the definition of such *behavioral invariants* and illustrate its use on simple use cases.

Key-words: CCSL, OCL, time constraints, semantics, MoC

* Université de Nice Sophia Antipolis

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

ECL: the Event Constraint Language, an Extension of OCL with Events

Résumé : Pas de résumé

Mots-clés : CCSL, OCL, contraintes temporelles, sémantique, MoC

n Domain Specific Modeling Languages, the metamodel captures the domain concepts and their relationships. The Object Constraint Language (OCL) can be used to add structural invariants or to describe pre and post conditions on method calls. However, both OCL and current metamodeling languages fail in specifying concurrency, causality relationships and timed behavior of the models.

In this paper, we present a model-driven approach to a formal and explicit specification of causal and timed relationships within models by extending the OCL. We describe our domain-specific modeling language dedicated to the definition of such *behavioral invariants* and illustrate its use on simple use cases.

1 Introduction

Models abstract away complex systems to focus on the relevant aspects only. An adequate abstractions should allow early validation/verification of the system. Consequently, the model and its underlying semantics are often specific and driven by the expected kinds of analyses and the nature of the properties to be verified. In many domains, models abstract the relative orderings of actions whose execution order are of prime importance: business process models, web services orchestrations, real-time/critical systems...

Since the mid 70s, computer science has used various kinds of models to abstract systems and perform analyses [1]. These models were first described by specific-purpose languages, now called Domain-Specific Languages (DSL). DSLs define relevant entities pertaining to the target domain. These entities are coupled by domain specialists with a formal behavioral semantics [2–7].

Nowadays, creating a DSL is well accepted and has been popularized by the tooling facilities provided by the MDE technologies. A metamodel defines concepts and relations of A specific language and can be used to generate powerful editors and programming interfaces. However, the help is mainly provided for the definition of structural models (*i.e.,* static models) but little help is given to ease the simulation/execution of the models. Among the approaches that provide tooling to specify the (meta)model dynamics [8–10], either the models can be executed/simulated only at the very last stage of the development process (once all model behaviors are encoded) or they fail to capture concurrency and loosely synchronized systems (like required for distributed or multi-core systems). Additionally, executable models often "implement" a specific solution while the problem could accept various ones. This is mainly related to the fine granularity of the description needed by existing tools to make the model executable.

An alternative to defining a DSL is the Unified Modeling Language (UML) [11], a general-purpose modeling language. Several UML profiles have been defined to extend the UML with stereotypes that denote relevant domain-specific entities. Applying a stereotype to a model element maps this model element to a domain entity, therefore enforcing the commonly accepted domain entity semantics. Some tooling can then be reused (*e.g.,* the graphical editors) but simulators/analysis tools still have to be developed according to the domain semantics.

What is missing in the current modeling approaches (DSL and UML-based) is an executable *semantic model* that captures within the same technological space and explicitly, the behavioral invariants to which the *domain model* must conform, *i.e.,* the constraints under which the model is amenable to analyses. Such behavioral constraints on the model should not over restrict the future model refinement/implementation and should be given from the very first stages of the development process. They should encapsulate, formally and explicitly, the behavioral semantics of a model for a specific domain but should not be a programming language that encodes a specific usage of the domain concepts. In this paper, we propose to add the possibility to specify behavioral invariants on a model by slightly extending the Object Constraint Language (OCL [9]). Models built with the approach are amenable to formal analysis and can be used either in simulation to bring confidence in the developed model or used as a *reference* model when transformations are performed to other analysis-specific formal languages. The proposed behavioral invariants focus on event ordering but can be complemented by the already existing features of OCL like pre/post

conditions.

The next section gives the rationale for this work. Section 3 discusses related approaches. Section 4 describes our proposal: a language dedicated to the definition of behavioral invariants, and its integration within OCL. The approach is illustrated by the specification of some behavioral invariants on a DSL and a UML model.

2 Motivations and prerequisites

Being able to constrain the relative execution ordering of actions in a model is of prime importance to conduct analysis. More precisely, depending on the execution constraints considered, some analyses are possible while others are not. The meaningful constraints as well as the associated analysis techniques have been largely studied in various domains [12–14]. The so-called Models of Computations (MoC) largely focus on the events while data are very often left aside This is the case, for instance, for synchronous approaches where the clock calculus is at the center of the process [15] that entirely depends on the synchrony hypothesis. This is also the case in real-time scheduling theory, where the hypotheses are quite different. Assuming regular execution of tasks (periodic, sporadic), the model can be analyzed with exact or approximate analytical approaches. Our goal is to provide a formalism to capture the hypotheses taken and the synchronization constraints as part of model driven engineering.

The Object Constraint Language (OCL [9]) is already largely used to capture constraints on (meta)models. The OCL is a declarative textual language that provides constraints and object query expressions on any MOF model or meta-model. These constraints can be either *invariants* or *pre/post conditions* on the behaviors. Invariants are used to specify structural restrictions on the instances of the model on which they are specified. The pre/post conditions are used to specify contracts on the state of the system before and after the execution of a specific behavior. The language intends to be simple enough to be used by any designer while being formally defined.

The OCL is said to be side effect free in the sense that the evaluation of constraints never modify the state of the system. However, it is possible in OCL to define new attributes and methods on (meta)classes. These definitions can then be used in the specification of constraints. For instance in the example 1, the number of children of a *Person* is stored in a new integer attribute, named *nb_children* for each Person object. The integer is initialized with the size of the *children* collection of the *Person* (meta)class:

Listing 1: Example of attribute definition in OCL

```
1 context Person
2  def: nb_children: Integer = self.children->size()
```

Such attributes can then be used in any constraints as if they were part of the (meta)model. It is also possible to define temporary variables to ease the reading of constraints. For instance, in the example 2, a temporary variable *maxChildrenNumber* is used in the invariant expression but cannot be used anywhere else. Additionally, in this constraint the attribute previously defined is used.

Listing 2: Example of invariant and temporary variable in OCL

```
1 context Person
2  inv notTooMuchChildren:
3   let maxChildrenNumber :Integer =
4           self.livingCountry.maximalNumberOfChildren in
5   self.nb_children <= maxChildrenNumber
```

Our motivation comes from the fact that some simple behavioral constraints cannot be expressed in OCL. Such constraints have to hold all along the execution and we call them *behavioral invariants*. A

first example of behavioral invariant could be: "for all components, output data ports are always produced simultaneously". A second different example can be: "For concept C1,it is only possible to call one method at a time, *i.e.,* methods cannot be called concurrently". This behavioral invariants may be applicable to every model of the domain and may be expressed at the metamodel level. These kinds of constraints cannot be specified in OCL while they are really important for the understanding of the metamodel behavior.

There also exists application-specific behavioral invariants. They can be related to performance issues (*e.g.,* deadline) or to specific synchronization mechanisms dictated by the application (consider vehicle indicators for instance). They can also reflect the good way to use a concept of the model. For instance, an application-specific behavioral invariant is: "The init() method must be called first, before any other method". If specified on a UML model, this last behavioral invariant can be expressed in OCL but by putting a precondition on every other method than init(). It makes this simple constraint difficult to understand and it reflects badly the goal of the constraint. Additionally, because the specification of this constraint in OCL has to use 'messages' it cannot be specified on a domain specific model.

In order to enable the specification of behavioral invariants, we want to slightly extend the OCL with the possibility to define event attributes and temporary events. These events specify interesting events in the (meta)model (*e.g.,* the call of a behavior, the reception of a data on a port, or any other (meta)model specific events). They may not be specified in the metamodel because they are used only to specify the behavioral semantics of the model and are not concepts of the modeled domain. Adding them directly in the model would over complexify the model for no reason. By making events explicit, it is possible to specify behavioral invariants on them (and particularly the ones under which the system is amenable to analysis). These behavioral invariants must be specified by using a formal but intuitive language to stay aligned with the OCL.

Amongst the benefits expected from such an extension of the OCL, we can cite, model simulation / animation at a high abstraction level (without precise specification of any data-dependent behavior), transformation checking, execution trace verification, generation of observers for the implementation, easier communication among domain specialists.

The next section describes related works that focus either on the extension of the OCL or on the explicit specification of behaviors at the metamodel level.

3 Related works

Our approach aims at extending the OCL so that the specification of behavioral invariants. Our goal is to specify the synchronization relationships amongst the events of a model. There exist various approaches that extends the OCL to enhance the behavioral constraint expressiveness. First we want to avoid ambiguity by stating that our goal is clearly not to put (some of) the temporal logic operators in the OCL; this has already been done in other approaches [16–18]. Adding temporal logics to the OCL is a good way to take benefits from a very expressive and formal language. However, it gets a rather low acceptance due to the complexity of temporal logics itself and the lack of abstraction mechanisms of such approaches. An alternative approach has been proposed by Flake *et al.* [19]. It consists in enhancing the OCL message-related concepts to add expressiveness on the pre/post condition constraints. The authors highlight the lack of expressiveness of the OCL behavioral constraints and proposes a clean and formal integration with the OCL semantics. Almost the same approach has been conducted in [20]. Two main drawbacks of these approaches have been identified. First, by focusing on the message-related concepts of OCL, they restrict the use of their approach to the UML. It is not possible to use any message-related concept on a DSL. It is also a problem of OCL itself, whose utilization is, nowadays, not intended to be limited to the UML. The second drawback of the approach is that, by using pre/post conditions to specify the synchronization, it is difficult to specify synchronizations that must be respected independently of a specific behavior. For

instance constraints due to the allocation of various components on a single processor or on the contrary due to the allocation on a distributed platform cannot be easily specified since they cannot be directly associated with a specific behavior or message call.

Cariou *et al.* [10] proposed an approach to specify the semantics of models by using the OCL. While we share with them the same goal, they chose to do it by using a contract-based approach (*i.e.,* by using the pre/post conditions). An interesting point shared with [21] is that they highlight the need to add some data used only to represent the dynamic state of the system. Consequently, they add specific methods to process these data and more precisely to specify (by using pre/post conditions) when the other behaviors can or cannot be called. For instance they add a *run_to_completion()* method to a state machine to specify contracts on the system state before and after the execution of the method. Their preconditions mainly rely on the concepts of UML events and it is not clear how it can be used in a DSL where such events do not exist. Also, because they use OCL without any extensions, they suffer from the same restrictions as the one identified by approaches that worked on OCL extensions: no message manipulation outside of the UML, poor synchronization capabilities, no way to specify concurrency explicitly and no operators to handle time.

All the proposed approaches use mechanisms based on pre/post conditions to specify the behavioral aspects of models. We believe that it is important to introduce explicit events and behavioral invariants to complement such approaches. Explicit events create a mandatory bridge between the static aspects of a metamodel and its dynamics (as stated in [21]). Behavioral invariants specify constraints that must be true independently of any method call. For instance, we want to be able to specify that two components allocated on a single CPU cannot be concurrent without giving any detail on the actual "behavior" of the CPU. The goal of behavioral invariants is to specify the acceptable (partial) orderings between the occurrences of the events and consequently has strong relations to the notion of MoC.

Finally, when speaking about making domain-specific executable models, we should mention KerMeta [8]. KerMeta is a metamodeling language, which allows structural and behavioral semantics to be defined directly within a metamodel. By using aspect-oriented modeling, KerMeta is well integrated in a modeling process. A classical use of KerMeta consists in specifying the behavioral semantics as aspects of a specific metamodel. The KerMeta framework is then able to weave the aspects into the Domain Specific Metamodel and each model that conforms to this metamodel is then an executable model. KerMeta has a great expressiveness that allows general descriptions and data manipulation but this expressiveness is achieved by using a java-like language. This language is too much operational and is used to implement a specific solution rather than to specify and declare a set of possible solutions. This way it is not possible to specify behavioral invariants directly. Such invariants would represent a large set of acceptable executions, like causality relationships for distributed systems. Obviously such invariants can be encoded in the KerMeta language but the invariants should be accessible as first-class citizens if analyses have to be conducted. Also KerMeta does not support timed invariants or concurrency operators (like periodic execution, parallel execution, deadline, etc.). However, the aspect-oriented feature of KerMeta is an elegant way to weave in a metamodel the data that represent the system states as well as the behaviors that process these data during the execution. For these reasons, we are currently studying a merge of both approaches: KerMeta for data manipulation, explicit events to refer to actions on these manipulations (call, start, suspend...) and the behavioral invariants to specify the orchestration of these manipulations in a non restrictive way.

4 Proposition

We want to be able to specify behavioral invariants on structural metamodels and models. To do so, we use the notion of event, an event being a totally ordered set of event occurrences. The behavioral invariants are then specific kinds of OCL expressions that restrict the (partial) ordering of the event occurrences in

the system. The language used to specify the behavioral invariants must be formal but easy to use by any designer. Additionally the expressiveness of the language must allow various kind of constraints to be specified, from loosely to very strong synchronizations between the events. Finally, to match the OCL style, this language must be declarative. We chose to adapt the Clock Constraint Specification Language (CCSL) to the OCL. CCSL, is a model-based declarative language initially introduced in the OMG MARTE profile [22]. It is used to specify relationships between *clocks* (*i.e.,* events). CCSL has a formal semantics [23] that defines valid partial order according to a given CCSL specification or rejects the specification when there are some contradictions. As a concrete syntax to handle *logical time* expressions, the CCSL has proved to be very expressive at different levels of abstraction.

Logical time has been successfully used in several domains. It was first introduced by Lamport to represent the execution of distributed systems [24]. It has then been extended and used in distributed systems to check the communication and causality path correctness [25]. During the same period, logical time has also been intensively used in synchronous languages [26,27] for its polychronous and multiform nature (*i.e.,* based on several time references). In the synchronous domain it has proved to be adaptable to various levels of description, from very flexible causal descriptions to precisely timed ones [28]. Actually, the notion of logical time is often used every day when a specific event is taken as a reference. For instance, consider the sentence "*Component 1 is executed twice as often as Component 2*". An event is then expressed relative to another one, that is used as a reference. No reference to physical time is given. However, given the execution occurrences of *Component 1*, the execution occurrences of *Component 2* can be deduced. Another example is: "*init()* must be called before any other methods". Once again, you schedule an event in time relatively to another one (this time in a very loosely synchronized way). This is the main idea of using logical time. In this context, physical time is a particular case of logical time where events generated by a physical clock are taken as references. Consequently, logical and multiform time allows considering not only the distance between two event occurrences (that would be expressed relative to the physical time) but also the relative ordering of event occurrences.

By taking advantage of logical time, CCSL has been used successfully in several domains ranging from the expression of the Synchronous Data-Flow (SDF) model of computation [29] to the specification of temporal constraints for EAST-ADL models [30] but also to reflect the behavioral impact of implementation on requirements [31] and so on [32–39].

The problem of CCSL is twofold. In the one hand it can only specify constraints in extension (*i.e.,* for each concept instances in a model) so that it is difficult to use on big models. In the other hand, CCSL only deals with constraints between the events and cannot take benefits from contract based approaches.

By adapting CCSL to the specification of behavioral invariants in OCL, we want to take benefits of both languages and we aim at providing a meaningful constraint language for MDE.

Before the presentation of the proposed language, we starts with an informal description of CCSL and its underlying mathematical model.

4.1 CCSL concepts and relations

In this paper, we describe only a small subset of CCSL, focusing on syntactic and semantic aspects sufficient to understand the examples given at the end of the paper. The syntax and the semantics of full CCSL is available in a technical report [23] and an overview is available at `http://timesquare.inria.fr`.

A simplified view of the CCSL metamodel is given in Figure 1[1]. A *TimeSystem* (aka CCSL specification) consists of a finite set of *Clocks*, and the parallel composition of a set of *ClockConstraints* (denoted | in the remainder). A clock is a strictly ordered set of instants, usually infinite. Two instants belonging to different clocks are possibly related by a causal or a temporal relationship. Causality is denoted $\preccurlyeq$. It can be refined into a temporal relationship, either a strict precedence (denoted $\prec$) or a coincidence

[1]The whole metamodel in ecore is available at http://timesquare.inria.fr/resources/metamodel/

(denoted $\equiv$). By combining such relationships, a time system is the specification of partially ordered sets of instants [40]. A clock constraint specifies generic relationships between (infinitely) many instants of the constrained clocks.

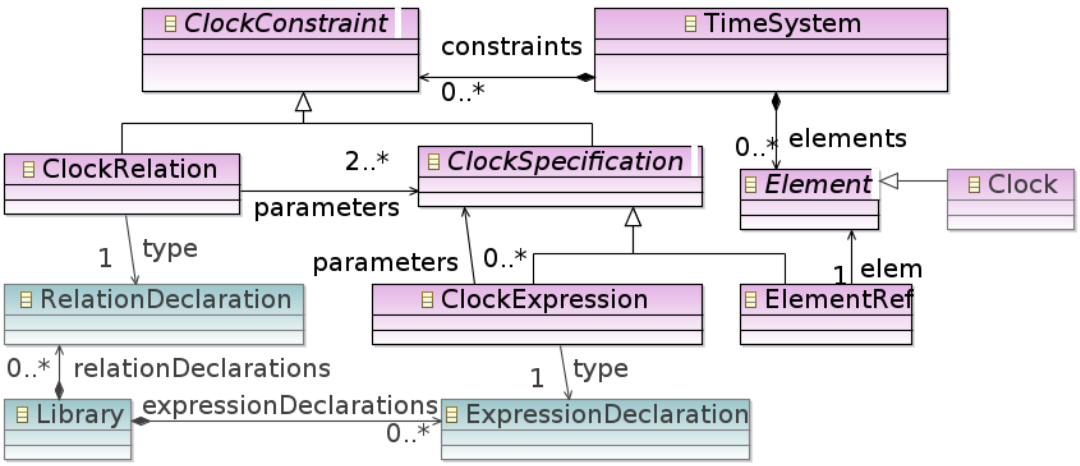


Figure 1: Simplified CCSL metamodel

A clock constraint is either a *ClockRelation* or a *ClockSpecification*. A clock relation mutually constrains its parameters according to its *type*. A clock specification is either a *ClockExpression* or a simple reference to an element (*e.g.,* a clock, an integer, etc) (*ElementRef*). A clock expression has parameters and specifies a new clock according to the *type* of expressions and of its parameters.

The types of the clock relations and expressions are defined in a *Library*. Low abstraction level constraints are defined in a kernel library. CCSL allows for the definition of user-defined constraints by combining existing relations imported from a set of kernel relations or from another user-defined library. It makes easier the use of complex constraints. However it makes the syntax quite verbose; especially for the binding of the formal parameters of the declaration to the actual ones of the constraint. This is not detailed here and we consider that the order of the parameters are used to bind them.

In this paper, we consider two types of clock relations, whose parameters are clocks references:

- the precedence (denoted $\prec$), which specifies that all instants of the first parameter occur before the corresponding instants of the second parameter.
- the coincidence (denoted $=$), which specifies that all instants of the first parameter are simultaneous/synchronous with the corresponding instants of the second parameter.

We also only use two clock expressions:

- Union expression (denoted $c1 \ + \ c2$), whose resulting clock ticks synchronously with each of the clocks provided as parameters.
- delayedFor expression (denoted $c \ \$ \ n \ \ on \ \ ref$), which takes two clocks (c and ref) as first and second parameters and an integer (n) as third parameter. The resulting clock ticks synchronously with the n^{th} tick of ref after each tick of c.

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

$$\mathcal{S} = \big(A \boxed{\prec} B \mid B \boxed{\prec} (A \$ 1 \ \mathit{on} \ A)\big) \tag{1}$$

This specification is slightly reformulated to facilitate the semantic explanations: we introduced one implicit clock C that evolves in coincidence with the delayedFor clock expression of the previous specification.

$$\mathcal{S} = A \boxed{\prec} B \mid \big(C \boxed{=} A \$ 1 \ \mathit{on} \ A\big) \mid B \boxed{\prec} C \tag{2}$$

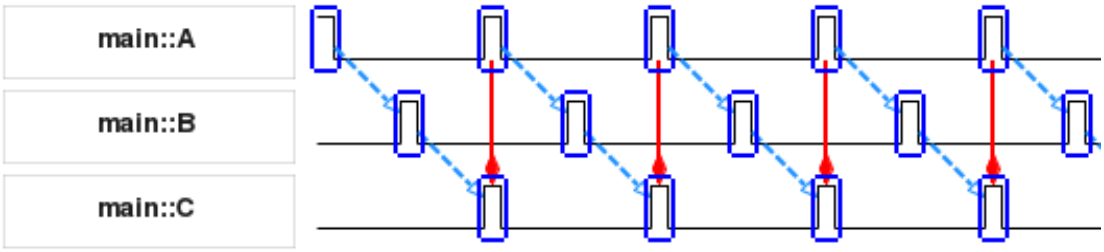


Figure 2: A timing diagram of the example in equation 2

Equation 2 is actually specifying that clock A and B alternate, as represented in the timing diagram Figure 2 provided by the TIMESQUARE environment [41] (blue dashed arrows represent precedences and red diamond lines represent coincidences). This kind of construction can be defined in a library so that the alternates relation can be directly used as if it was part of the language (without corrupting the formal semantics). The corresponding *RelationDeclaration* is then:

$$\begin{aligned} &\textsf{def}\ \mathit{Alternates}(\textsf{clock}\ \mathit{leftClock},\ \textsf{clock}\ \mathit{rightClock}) \triangleq \\ &\quad \mathit{leftClock} \boxed{\prec} \mathit{rightClock} \mid \\ &\quad C \boxed{=} \mathit{leftClock} \$ 1 \ \mathit{on} \ \mathit{leftClock} \mid \\ &\quad \mathit{rightClock} \boxed{\prec} C) \end{aligned} \tag{3}$$

This is the goal of the libraries to provide domain-specific constraints, constructed by semantic specialists but used by (meta)model designers. In this section, the concrete syntax is not the one used in the tool but has been used in the semantic definition because it is more concise than the actual one. The actual concrete syntax definition is available here: `http://timesquare.inria.fr/index.php?slab=extendedccsl-grammar`

4.2 CCSL operational semantics

This paragraph gives the flavor of the CCSL semantics without falling into the details. Please refer to [23] for a more precise definition. 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 clocks that can/must "tick" to represent the evolution of the system. This set is computed according to the specified 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.

4.3 extension of the OCL metamodel: ECL

Our goal is to allow the manipulation of logical time according to the semantics defined by CCSL but in an OCL-like manner (*i.e.,* possibly at the metamodel level and by intension instead of by extension). A first solution consists in creating a specific OCL library that extends the existing OCL standard library with the notion of event and some predefined constraints on them. However, this method does not allow the creation of domain-specific behavioral invariants (*i.e.,* like the constraints of a CCSL library). To avoid repulsing users with difficult specifications, we extended the OCL metamodel itself so that the use of high abstraction level constraints is possible. We named this metamodel ECL for Event Constraint Language, however, because any other constraints from OCL are accepted by the metamodel, the name should surely be reconsidered.

ECL added three new possibilities to the OCL language:

1. The definition of Event; in the same way than the already existing definition of new attributes or temporary attributes
2. The initialization of an event from already existing event(s) and model attributes
3. The specification of relations between the events in the model; in the same way than already existing relations between other attributes.

There exist various implementation of the OCL specification. We chose to extend the implementation provided in the eclipse MDT[2]. This implementation supports several metamodel depending on whether we consider the "complete OCL" or only fragments specified, for instance, in UML constraints. To avoid being intrusive in every metamodel, we decided to create a new metamodel extending the complete OCL metamodel. Our extension is shown in Figure 3. The yellow metaclasses are the ones from the existing OCL metamodels, the ones in Red are new metaclasses, and the ones in blue are from the CCSL metamodel. Actually only the relation and expression declarations are imported from the CCSL metamodel in order to take benefits from the semantics of kernel relations and expressions defined in the MARTE profile as well as from user-defined library.

An *ECLDocument* is the root of the specification. An ECLDocument is a CompleteOCLDocument. Only specific imports are added to enable references to relation and expression libraries. Imports are specified by an *ImportStatement*. It contains the path of a library file whose RelationDeclaration and ExpressionDeclaration are used by the relations and expressions of the ECL specification. To allow the addition of events to a (meta)model, we have added the new type *EventType*. This type can be used in a classical OCL definition (like in listing 1) as well as in temporary variables (like in listing 2). Note that it does not, in anyway, make side effects since these values/methods are not used in the model itself, they are just used by the ECL specification. Additionally, these attributes can be initialized and read but cannot be assigned, as requested by the OCL specification. An *EventType* can be parameterized by a *referredElement* used to specify the (meta)model element the event attribute is associated with. This referred element is an *ExpCS*, *i.e.,* an OCL expression, so that it can be a navigating expression, querying a specific modeling element (e.g. $self$, or $self.incoming->select(i|i.stuff <> null)->first()$). When specifying a new attribute of type EventType, the event has to be initialized. The *EventLiteralExp* is a way to initialize the event attribute. It possesses a value, which specifies the role of the event with regards to its referred element. This role is an enumeration (*EventKind*). A *Relation* is an OCL expression whose goal is exactly the same than the CCSL relation, i.e. constraining the relative order between the occurrences of the events in its parameter list. The type of a relation is a *RelationDeclaration*. The parameters are also OCL expressions so that they can be specified by a navigating expression, querying a specific event or a specific model attribute depending on the type of the parameters specified in its type. Note that, a relation declaration has an ordered set of variables. The first parameter in a relation is then

[2] `http://wiki.eclipse.org/MDT/OCL`

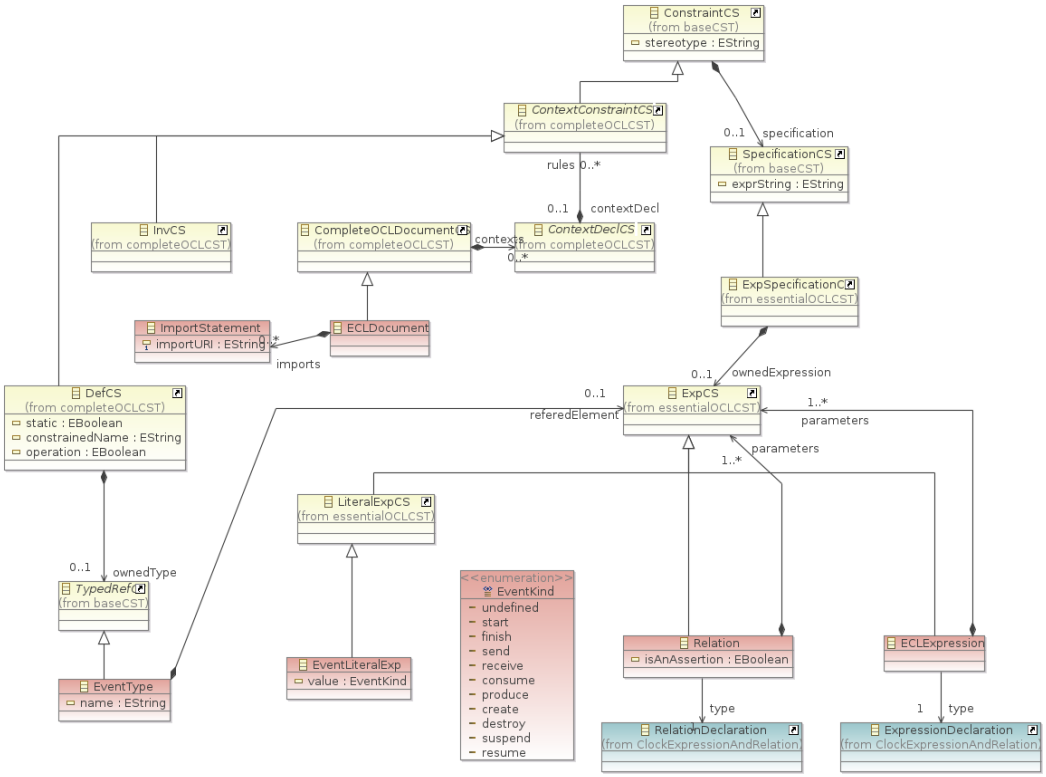


Figure 3: the extension of the OCL metamodel to allow the specification of behavioral invariants

bound to the first variable in its type, the second with the second and so on... In OCL, one can use a let expression (*LetExpCS*) to create a temporary variable from other model elements or variables, and used it in a constraint. When the constraint is a relation on events, one can need to consider an event which results from the relative evolution of other events as well as on other parameters (for instance the union of a set of events). The *ECLExpression* can be used in a let expression to specify a temporary event. Similarly to relations, the type of an ECL expression is an *ExpressionDeclaration*.

The result of the metamodel extension is the possibility to specify behavioral invariants as well as any other OCL constraints. Of course to make it usable, we have also modified the concrete syntax. The implementation of OCL in eclipse uses XText[3]. We extended the syntax with three main keywords: Event, Relation and Expression. All the other introduced keywords are used to specify the event kind enumeration. Examples of OCL specifications with behavioral invariants are provided in section 4.5. The next section details the use of such behavioral invariants.

4.4 ECL tooling and usages

The first goal of an OCL specification is to improve the specification of the model-based specification. The invariants specify structural constraints on models. Many tools provide a checker to validate these invariants on a specific model. Considering the pre and post conditions, they require another kind of tools because they refer to the execution of the model. They can be used in different manners according to the context. A first use of them consists in translating the pre post conditions to their equivalent in a programming language when the targeted language supports such contracts (*e.g.,* for KerMeta and the Eiffel languages). A second use consists in using such contracts as a reference for model manipulations (*e.g.,* transformations, execution, ...) where the manipulation is said to be correct when it respects such contracts [10, 42].

In the same way than pre and post conditions, we will use behavioral invariants in different ways according to the context. The main goal is to take benefits of the specification in order to provide confidence either in the developed model or in its manipulation/implementation. We have seen that the extension of the OCL language is based on the semantics of the CCSL language. As far as we know, there exists two frameworks able to process a CCSL specification to take advantage of it [41, 43]. To take benefits of these tools, we generated a CCSL specification according to the ECL specification and an input model. By doing so, we can exploit the possibilities provided by these tools depending on the context:

- obtaining a satisfying trace model linked with the model (in any cases)
- generation of timing diagrams (in any cases)
- graphical animation of the model (if modeled in the Papyrus tool)
- generation of observers to have a run-time checking of the implementation (for esterel and VHDL languages)
- checking of some execution traces according to the specification (if the trace is based on the open trace format)
- exhaustive simulation of synchronization properties (when the state space is finite by translation to timed petri nets)

In order to create the corresponding CCSL specification, we realized an high order transformation whose inputs are the ECL constraints and the domain-specific metamodel and whose result is a transformation from a domain-specific model to a CCSL specification (see Fig. 4). This last transformation

[3] `http://www.eclipse.org/Xtext/`

is reusable for all models that conform to the metamodel used in input of the high order transformation. From a technological point of view, the first transformation has been implemented using acceleo[4]. It generates a QVTO transformation[5] that can be used to automatically create CCSL specifications.

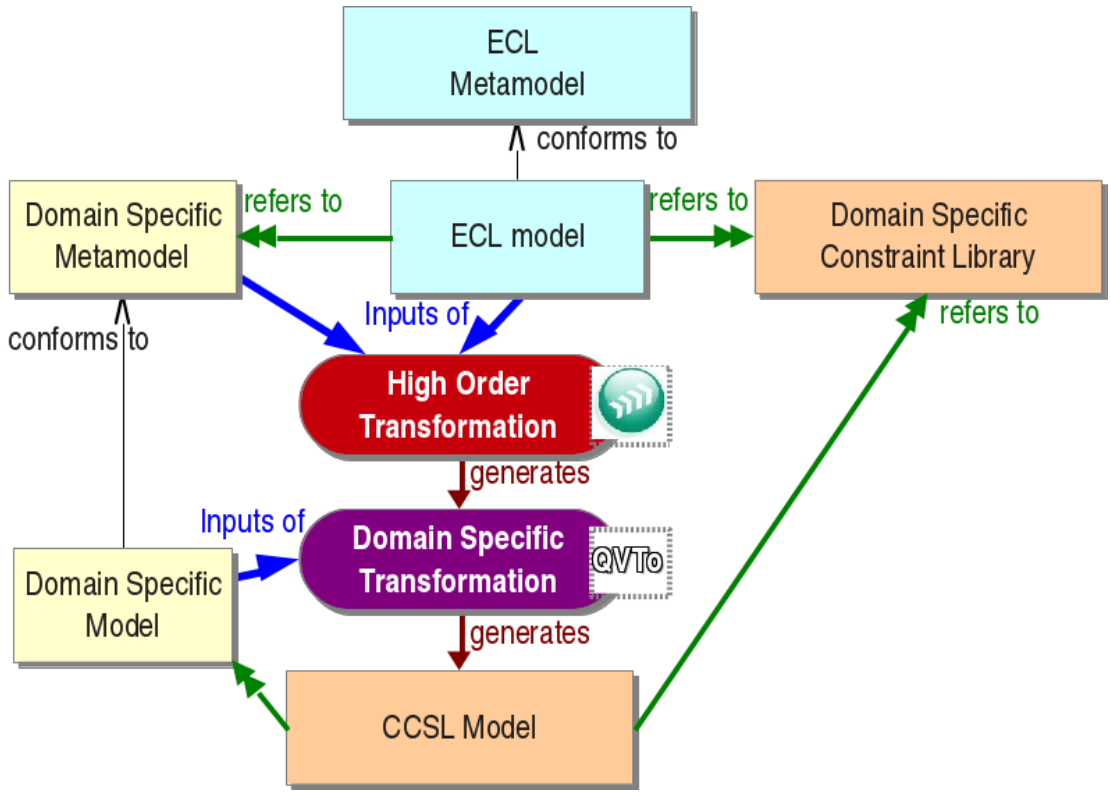


Figure 4: models and transformations to take benefits from CCSL tooling

4.5 Examples of behavioral invariant specification at the metamodel level

4.5.1 Simple Component Model

This first example explains how to specify some behavioral invariants on a specific metamodel, like the component metamodel introduced in Figure 5. We specify the behavioral invariants by using relation declaration whose semantics has been introduced in section 4.1. In this section, we use the textual syntax of ECL and relation declaration are referred by their textual name instead of the mathematical syntax introduced in Section 4.1.

The first behavioral invariant we want to specify is taken from section 2. It states that "for a *Component*, it is only possible to call one behavior at a time". In other words, if one behavior has already started, no other behaviors can be started while the first one has not finished.

To specify this behavioral invariant, the first step consists in choosing the model elements that are relevant from a behavioral constraints point of view. In this example, both *Component* (line 8 of Figure 6)

[4] `http://www.eclipse.org/acceleo/`
[5] `http://wiki.eclipse.org/M2M/Operational_QVT_Language_(QVTO)`

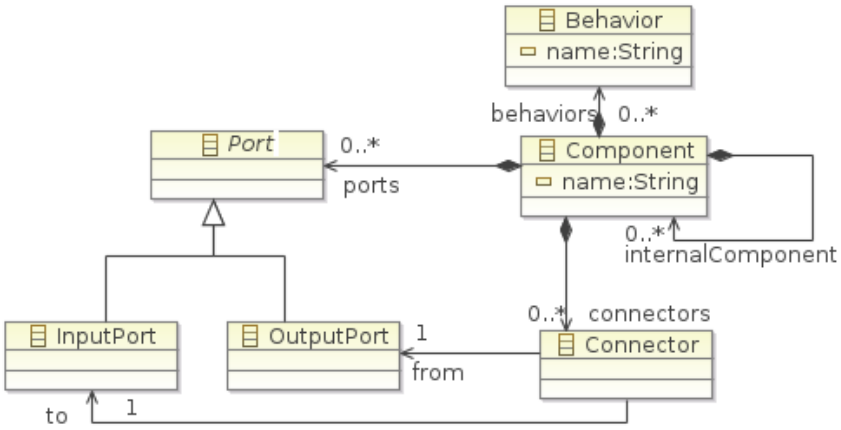


Figure 5: A sketchy and partial component metamodel

and *Behavior* (line 15 of Figure 6) are used. In a second step, we define, for these elements, the events used in the invariant. In our case, the start and the end of a behavior are used. We consequently define two event attributes in the context of a Behavior (line 9 and 10 of Figure 6). It is now necessary to constrain these events. We first specify that the start of a behavior always precedes its stop. This is specified by the *startPrecedesStop* behavioral invariant (line 12,13 of Figure 6), that uses the *Precedes* relation declaration from a CCSL library. Now, because we do not want more than one behavior to execute at a time, it means that they cannot start simultaneously. This is the goal of the second behavioral invariant, named *notSimultaneousStarts* on the line 16 and 17 of Figure 6. This invariant specifies an *Exclusion* relation on a collection of events. It means that all events in the collection are exclusive with each other. Note that this is a simplification of the CCSL syntax where exclusion on a set must be specified for each pair of clocks in the set. Finally, in the last behavioral invariant we specified that no behavior can start if another one has already started. Consequently, it is impossible to observe two consecutive start events without a stop. This is done by using temporary events (line 20 and 21). These events, named *allStart* and *allStop* are respectively defined by the *Union* of all start and stop events of the behavior of a component. These temporary events are then constrained to *Alternates*.

This example has been used to present the idea behind our extension of the OCL. In the next subsection, we present a more realistic example where an activity diagram is constrained so that it follows the execution rules of Synchronous Data Flow (SDF [44]) graphs, a formal model suitable to analysis.

4.5.2 Synchronous Data Flow

The goal of this section is not to explain the constraint needed on a UML activity model to give it the semantics of SDF. The goal is to show how, by using a high level library developed by experts, a model behavioral semantics can be formally specified. The library internal and details about its specification can be found in [29]. The result of [29] is that, it is possible to construct a *RelationDeclaration*, whose uses on a structurally constrained activity diagram, gives the behavioral semantics of SDF. The structural constraints put on the activity diagram enforce that the control flow between two actions go through a join node. The initial mark (delay) of the *arc* can then be given by using an initial node associated with this join node. An example of such activity model is provided in Figure 7.

The high abstraction level relation developed in [29] has the following signature: def Arc(int $delay$, event $sourceExec$
In this relation, the *delay* integer represents the initial number of data on the arc. The *sourceExec* event

SimpleComponent.ecore SLE2012.ecl

```
1  import './SimpleComponent.ecore'
2
3  ECLimport  "./kernel.ccslLib"
4  ECLimport "./CCSL.ccslLib"
5
6  package simplecomponent
7
8      context Behavior
9          def : start : Event(self) = StartEvent
10         def : finish : Event(self) = StopEvent
11
12         inv startPrecedesStop:
13             Relation Precedes(self.start, self.finish)
14
15     context Component
16         inv notSimultaneousStarts:
17             Relation Exclusion(self.behaviors.start)
18
19         inv noMultipleCallOfInternalBehavior:
20         let allStart : Event = Expression Union(self.behaviors.start) in
21         let allStop : Event = Expression Union(self.behaviors.stop) in
22             Relation Alternates(allStart,allStop)
23
24 endpackage
```

Figure 6: a simple ECL specification in the ECL editor

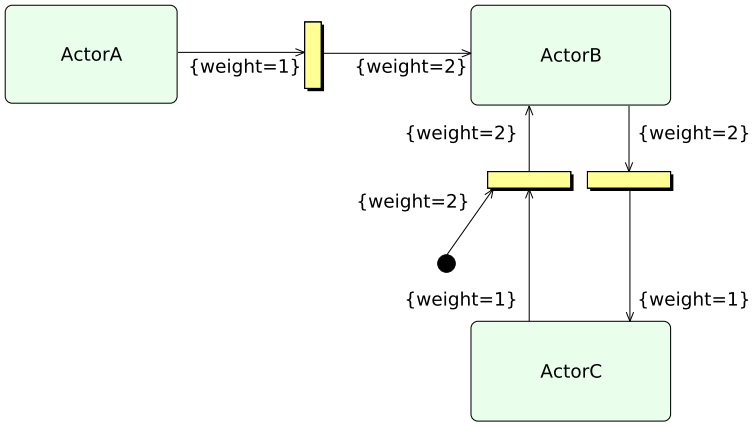


Figure 7: a UML activity diagram with join node between actions

represents the atomic execution of the node in input of the join node. Similarly *targetExec* represents the atomic execution of the node in output of the join node. The two integers *in* and *out* respectively represent the number of data consumed on a specific arc and the number of data produced on a specific arc when an action executes.

The behavioral invariant is then call after retrieving the parameters of the relation by using the query operators provided by OCL. The result of the specification is depicted in Figure 8. On this screenshot, only the highlighted parts are from our extension; the remaining part of the specification is pure OCL.

```
1  import 'http://www.eclipse.org/uml2/2.1.0/UML'
2
3  EClimport "SDF.ccslLib"
4
5  package uml
6
7      context Action
8      def : doExecution: Event(self) = GenericEvent
9
10     context JoinNode
11     inv ArcConstraint:
12         let sourceActor : Event = self.incoming->select(e |
13                 e.source.oclIsKindOf(Action)
14         )->asOrderedSet()->first().source.oclAsType(Action).doExecution in
15         let targetActor : Event =
16             self.outgoing->asOrderedSet()->first().target.oclAsType(Action).doExecution in
17         let outWeight : Integer =
18             self.incoming->select( e| e.source.oclIsKindOf(Action)).weight
19                                                 .oclAsType(LiteralInteger).value->sum() in
20         let delay : Integer =
21             self.incoming->select( e| e.source.oclIsKindOf(InitialNode)).weight->sum() in
22         let inWeight : Integer =
23             self.outgoing.weight.oclAsType(LiteralInteger).value in
24         Relation Arc(delay, sourceActor, outWeight, targetActor, inWeight)
25
26 endpackage
```

Figure 8: an ECL specification to constrain an activity model to conform to SDF semantics

The result of the high order transformation (the QVTo transformation) as well as the CCSL specification resulting from the transformation can be consulted at `http://timesquare.inria.fr/examples/ECL/`. If the first benefit is to provide a concise specification of the behavioral invariants integrated within a standard language, the second benefit comes from the ease to read and understand the specification compared to the associated QVTo transformation.

4.6 Critics of the approach and its tooling

The approach is a first step towards a specification of causality relationships, concurrency and logical time behavior in an MDE approach. It is a fact that creating behavioral invariants requires a good knowledge and experience of the CCSL introduced in the MARTE profile. The same kind of knowledge and experience are also required to provide a good metamodel. We believe that the possible use of high abstraction level libraries and the use of an existing language is a way to reduce the gap between the people that create formal models and the ones that develop metamodels. Our approach aims at making a separation between the abstract syntax of a language (the metamodel); its behavioral invariants (*i.e.,* the specification of the model of computation); and the implementation of a specific model that must respect both structural and behavioral constraints. The CCSL semantics was successful in specifying various kinds of behavioral invariants on models (see the beginning of section 4) but more experiments are required

to completely validate the proposition. These experiments have been a motivation for the tooling of the approach. However there still exist some limitations on the tooling of the approach.

First, we extended the OCL implementation of the eclipse project. This implementation uses various technologies that are still evolving. It makes it difficult for our editor and transformation to be up to date with the various releases of these technologies. We consequently wait for the release of the eclipse Juno before proposing any update site of our tooling.

Also, to exploit tools that process CCSL specifications, we developed an high order transformation that takes an ECL file as an entry. The main limit comes from this transformation. There often exist various ways to specify the same OCL constraint. In our transformation, only a part of these constructions are supported. For instance, it is not possible to use more than one relation in a single behavioral invariant. It is also not supported if in a single invariant classical OCL Boolean expressions are put in conjunction with an ECL relation. These kinds of limitations are not very inconvenient, and we tried to support all part of the language needed to drive new experiments. Additionally, by using the TIMESQUARE software environment, it is possible to have convenient feedback on the resulting semantics, making easier the development of new semantics for our models (`http://timesquare.inria.fr/index.php?slab=screenshots`).

5 Conclusion

This paper presents an extension of the OCL language. This extension allows the explicit specification of relevant events of a model. It also allows the specification of behavioral invariants. Behavioral invariants is a way to specify constraints on the partial ordering of the event occurrences in a model. The goal is to specify the causality relationships, the concurrency and the timed behavior of a model, so that domain-specific analyses can be conducted. Such behavioral descriptions can then be used in various scenarios ranging from model animation to generation of observers at runtime. The extension semantics is based on the *Clock Constraint Specification Language* (CCSL), a model-based declarative and formal language.

CCSL has been chosen because it supports, by using polychronous logical time, a timed causality model that brings consistency between interactions of the model element events. CCSL can be used to specify (not program) the expected behavior of the whole model. The extension, named ECL, proposed in this paper aims at complementing (not replacing) other features of the OCL language like the use of pre/post conditions.

ECL simplifies the CCSL language while keeping its formal semantics and allows the specification of behavioral invariants at the metamodel level. This specification uses relations and expressions defined in external libraries, by semantic experts. One goal was to keep OCL simple to use by non formal designer while allowing a formal specification of the causalities and concurrency in a system.

A prototype has been developed to allow the reuse of existing analysis tools like TIMESQUARE. The tool allows specific formal model to be created, according to the ECL specification. This helped in demonstrating the benefits of the approach, either to obtain early simulation of models, or to use another output of the simulation tool (like the verification of execution traces).

There are two main ongoing investigations to extend this work. A short-term objective is to provide our extension and its tooling as an update site in order to get feedback from the community. The direct results of this short-term objective is a study about the formal specification of heterogeneous systems; where a single model uses different relation and expression libraries (*i.e.,* heterogeneous modeling). Another future work consists in studying extensions of the language like the capacity to use the notion of mode to specify behavioral invariants that are dynamically enabled or not according to the value of data in the model.

References

[1] F. DeRemer, H. Kron, Programming-in-the-large versus programming-in-the-small, in: Proc. of the Int. Conf. on Reliable software, ACM Press, 1975, pp. 114–121. doi:10.1145/800027.808431.

[2] S. Faucou, A.-M. Déplanche, Y. Trinquet, An ADL centric approach for the formal design of real time systems, In Architecture description language, IFIP (2004) 67–82.

[3] EAST-EEA project, Definition of language for automotive embedded electronic architecture, Version 1.02.

[4] S. Vestal, MetaH user's manual - version 1.27 (1998).

[5] P. Feiler, B. Lewis, S. Vestal, The SAE avionic architecture description language (AADL) standard: A basis for model-based architecture driven embedded systems engineering, RTAS Workshop on Model-Driven Embedded Systems.

[6] R. Allen, A formal approach to software architecture, Ph.D. thesis, Carnegie Mellon, School of Computer Science, cMU Technical Report CMU-CS-97-144. (January 1997).

[7] J.-M. Farines, B. Berthomieu, J.-P. Bodeveix, P. Dissaux, P. Farail, M. Filali, P. Gaufillet, H. Hafidi, J.-L. Lambert, P. Michel, F. Vernadat, The Cotre project: rigorous development for real time systems in Avionics , in: WRTP'03 - Work. on real-time programming, Logow(Pologne), IEEE, 2003, pp. 51–56.

[8] P.-A. Muller, F. Fleurey, J.-M. Jézéquel, Weaving executability into object-oriented meta-languages, in: L. C. Briand, C. Williams (Eds.), MoDELS, Vol. 3713 of Lecture Notes in Computer Science, Springer, 2005, pp. 264–278.

[9] OMG, Object Constraint Language, Version 2.3.1 formal/2012-01-01 (January 2012).

[10] E. Cariou, C. Ballagny, A. Feugas, F. Barbier, Contracts for model execution verification, in: Modelling–Foundation and Applications: 7th European Conference, ECMFA 2011, Birmingham, UK, June 6-9, 2011, Proceedings, Vol. 6698, Springer, 2011, pp. 3–18.

[11] OMG, Unified Modeling Language, Superstructure, Version 2.1.2 formal/2007-11-02 (November 2007).

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

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

[14] A. Jantsch, Modeling embedded systems and SoC's: concurrency and time in models of computation, Morgan Kaufmann (an imprint of elsevier science), 2004.

[15] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, R. D. Simone, The synchronous languages twelve years later, in: Proceedings of the IEEE, 2003, pp. 64–83.

[16] M. Cengarle, A. Knapp, Towards ocl/rt, in: L.-H. Eriksson, P. Lindsay (Eds.), FME 2002:Formal MethodsâGetting IT Right, Vol. 2391 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2002, pp. 295–311.
URL http://dx.doi.org/10.1007/3-540-45614-7_22

[17] J. Bradfield, J. Filipe, P. Stevens, Enriching ocl using observational mu-calculus, Fundamental Approaches to Software Engineering (2002) 50–76.

[18] P. Ziemann, M. Gogolla, Ocl extended with temporal logic, in: Perspectives of System Informatics, Springer, 2003, pp. 617–633.

[19] S. Flake, Enhancing the message concept of the object constraint language, in: In Sixteenth International Conference on Software Engineering and Knowledge Engineering (SEKE 2004), Citeseer, 2004, pp. 161–166.

[20] M. Kyas, F. de Boer, On message specifications in ocl, in: UML 2003 Workshop on Compositional Verification of UML Models, 2003.

[21] X. Crégut, B. Combemale, M. Pantel, R. Faudoux, J. Pavei, Generative technologies for model animation in the topcased platform, in: Modelling Foundations and Applications: 6th European Conference, ECMFA 2010, Paris, France, June 15-18, 2010, Proceedings, Vol. 6138, Springer-Verlag New York Inc, 2010, pp. 90–103.

[22] The ProMARTE Consortium, UML Profile for MARTE, beta 2, Object Management Group, OMG document number: ptc/08-06-08 (June 2008).

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

[24] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Communications of the ACM 21 (7) (1978) 558–565.

[25] C. Fidge, Logical time in distributed computing systems, Computer 24 (8) (2002) 28–33.

[26] A. Benveniste, P. Le Guernic, C. Jacquemot, Synchronous programming with events and relations: the SIGNAL language and its semantics, Sci. Comput. Program. 16 (2) (1991) 103–149.

[27] G. Berry, The foundations of Esterel, Proof, Language and Interaction: Essays in Honour of Robin Milner (2000) 425–454.

[28] F. Boussinot, R. De Simone, The ESTEREL language, Proceedings of the IEEE 79 (9) (2002) 1293–1304.

[29] F. Mallet, J. DeAntoni, C. André, R. De Simone, The Clock Constraint Specification Language for building timed causality models, Innovations in Systems and Software Engineering 6 (1-2) (2010) 99–106. doi:10.1007/s11334-009-0109-0.
URL http://hal.inria.fr/inria-00464894/en

[30] F. Mallet, M.-A. Peraldi-Frati, C. AndrÃ©, Marte CCSL to execute East-ADL timing requirements, in: Int. Symp. on Object/component/service-oriented Real-time distributed Computing (ISORC'09), IEEE Computer Press, Japan, Tokyo, 2009, pp. 249–253.

[31] M.-A. Peraldi-Frati, J. DeAntoni, Scheduling Multi Clock Real Time Systems: From Requirements to Implementation, in: International Symposium on Object/Component/Service-oriented Real-time Distributed Computing, IEEE computer society, Newport Beach, United States, 2011, p. 50; 57, newPort Beach. doi:10.1109/ISORC.2011.16.
URL http://hal.inria.fr/inria-00586851/en

[32] F. Mallet, C. André, J. DeAntoni, Executing AADL models with UML/Marte, in: Int. Conf. Engineering of Complex Computer Systems - ICECCS'09, Potsdam, Germany, 2009, pp. pp. 371–376. doi:10.1109/ICECCS.2009.10.
URL http://hal.inria.fr/inria-00416592/en

[33] C. Glitia, J. DeAntoni, F. Mallet, Logical time @ work: Capturing data dependencies and platform constraints, in: T. J. J. Kazmierski, A. Morawiec (Eds.), System Specification and Design Languages, Vol. 106 of Lecture Notes in Electrical Engineering, Springer New York, 2012, pp. 223–238, 10.1007/978-1-4614-1427-8_14.
URL http://dx.doi.org/10.1007/978-1-4614-1427-8_14

[34] J. DeAntoni, F. Mallet, F. Thomas, G. Reydet, J.-P. Babau, C. Mraidha, L. Gauthier, L. Rioux, N. Sordon, RT-simex: retro-analysis of execution traces, in: K. J. S. Gruia-Catalin Roman (Ed.), SIGSOFT FSE, Vol. ISBN 978-1-60558-791-2, Santa Fe, United States, 2010, pp. 377–378. doi:10.1145/1882291.1882357.
URL http://hal.inria.fr/inria-00587116/en

[35] K. Garcés, J. DeAntoni, F. Mallet, A Model-Based Approach for Reconciliation of Polychronous Execution Traces, in: SEAA 2011 - 37th EUROMICRO Conference on Software Engineering and Advanced Applications, IEEE, Oulu, Finland, 2011.
URL http://hal.inria.fr/inria-00597981/en

[36] J.-F. Le Tallec, J. DeAntoni, R. De Simone, B. Ferrero, F. Mallet, L. Maillet-Contoz, Combining SystemC, IP-XACT and UML/MARTE in model-based SoC design, in: Workshop on Model Based Engineering for Embedded Systems Design (M-BED 2011), Grenoble, France, 2011.
URL http://hal.inria.fr/inria-00601840/en

[37] C. André, J. DeAntoni, F. Mallet, R. De Simone, The Time Model of Logical Clocks available in the OMG MARTE profile, in: S. K. Shukla, J.-P. Talpin (Eds.), Synthesis of Embedded Software: Frameworks and Methodologies for Correctness by Construction, Springer Science+Business Media, LLC 2010, 2010, p. 28, chapter 7.
URL http://hal.inria.fr/inria-00495664/en

[38] R. Gascon, F. Mallet, J. DeAntoni, Logical time and temporal logics: comparing UML MARTE/CCSL and PSL, in: 18th International Symposium on Temporal Representation and Reasoning (TIME'11), Lubeck, Germany, 2011, pp. 141–148. doi:10.1109/TIME.2011.10.
URL http://hal.inria.fr/hal-00597086/en

[39] H. Yu, J.-P. Talpin, L. Besnard, T. Gautier, F. Mallet, C. André and, R. de Simone, Polychronous analysis of timing constraints in uml marte, in: Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2010 13th IEEE International Symposium on, 2010, pp. 145 –151. doi:10.1109/ISORCW.2010.10.

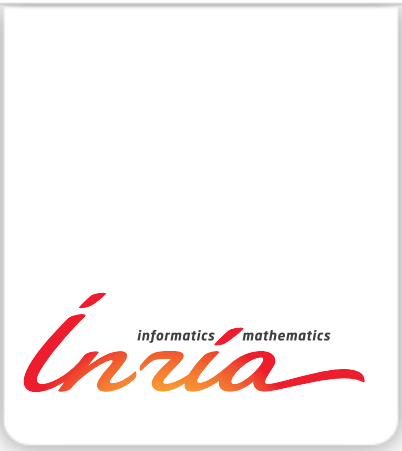
[40] C. André, F. Mallet, R. de Simone, Modeling time(s), in: G. Engels, B. Opdyke, D. Schmidt, F. Weil (Eds.), MoDELS, Vol. 4735 of Lecture Notes in Computer Science, Springer, 2007, pp. 559–573.

[41] J. Deantoni, F. Mallet, TimeSquare: Treat your Models with Logical Time, in: S. N. Carlo A. Furia (Ed.), TOOLS - 50th International Conference on Objects, Models, Components, Patterns - 2012, Vol. 7304 of Lecture Notes in Computer Science - LNCS, Czech Technical University in Prague, in co-operation with ETH Zurich, Springer, Prague, Tchèque, République, 2012, pp. 34–41. doi:10.1007/978-3-642-30561-0_4.
URL http://hal.inria.fr/hal-00688590

[42] E. Cariou, R. Marvie, L. Seinturier, L. Duchien, Ocl for the specification of model transformation contracts, in: OCL and Model Driven Engineering, UML 2004 Conference Workshop, Vol. 12, 2004, pp. 69–83.

[43] N. Ge, M. Pantel, Verification of Synchronization-Related Properties for UML-MARTE RTES Models with a Set of Time Constraints Dedicated Formal Semantic, in: 7th IEEE International Workshop on UML and AADL , 2012.
URL `http://hal.archives-ouvertes.fr/hal-00677925`

[44] E. A. Lee, D. G. Messerschmitt, Static scheduling of synchronous data flow programs for digital signal processing, IEEE Trans. Computers 36 (1) (1987) 24–35.



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