Extending OCL with Logical Time Behavioral Invariants

Julien DeAntoni and Frederic Mallet

Aoste Team-Project
Université Nice Sophia Antipolis
I3S - UMR CNRS 7271
INRIA Sophia Antipolis Méditerranée
2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex- France

Abstract. In the domain of real-time and embedded systems, models are often used for analysis and validation of the behavioral aspects of the system. The models must comply with a meta-model that mainly defines the concepts and composition rules (*i.e.,* the state semantics). The Object Constraint Language (OCL) can be used to add structural invariants or to describe pre and post conditions. However, OCL fails in specifying concurrency, causalities and timed behavior.
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 a simple component model.

We describe our domain-specific modeling language dedicated to the definition of such behavioral invariants and illustrate its use on a simple component model.

1 Introduction

Models abstract away complex systems to focus on the relevant aspects only. In the domain of Distributed Real-Time and Embedded Systems (DRES), 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.

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. Considering DRES and their demand for analysis, these entities are coupled by domain specialists with a formal behavioral semantics [2,3,4,5,6,7]. An alternative to defining a DSL is the Unified Modeling Language (UML) [8], 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.

Nowadays, creating a DSL is well accepted and has been popularized by the tooling facilities provided by the MDE technologies. The metamodel defines concepts of the 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 [9,10,11], either the models can be executed/simulated only at the very last stage of the development process (once all models 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.

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. 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 extending the Object Constraint Language (OCL). 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 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 and discusses related approaches. The third section 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 simple component model.

2 Motivations and overview

To better highlight the problem, we present in this section a little example where, for a single set of (structural) elements, the "behavioral understanding" is different depending on the domain and sensitivity of the reader.

Let us consider the component model depicted in Figure 2. Without knowledge of the application domain, it is pretty much impossible to say anything on the concurrency of the component execution. Does *Component_2* execute as soon as there is an interaction on one of its ports following an event-based execution policy ? Or is it executed periodically according to a time-triggered policy ? One could say that the behavior of each component is dictated by a state machine or another behavior description (like code for instance) inside the component. In this case, there is one main issue: depending on the behavior, some analyses can be conducted while others cannot. Depending on the domain, it is usually needed to restrict the possible behavior for every model of a specific domain so as to enable domain-specific analyses. Usually such restrictions do not concern data manipulations but rather the relative scheduling of the events in the systems. For instance, the synchronous approaches, by restricting the synchroniza-

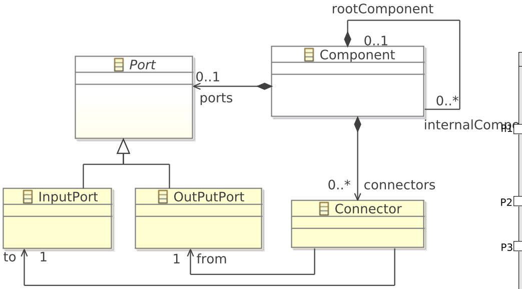


Fig. 1. A sketchy and partial component metamodel

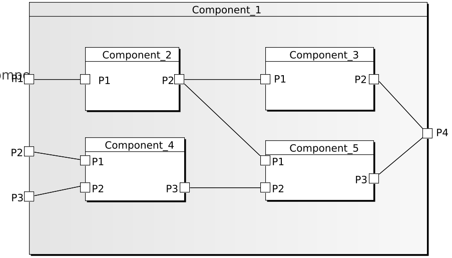


Fig. 2. A simple component model that conforms to its metamodel

tions of event to be all synchronous achieved lots of meaningful analyses and optimizations [12]. Other examples have been largely detailed in studies on model of computations [13,14,15]. In any case, the common goal is to restrict the possible behavior of a system in order to make it amenable to domain-specific verification and validation.

In the MDE domain, OCL (Object Constraint Language [10]) is a natural candidate that allows adding *invariants* on the model as well as *pre/post conditions* on its owned behaviors. Invariants are used to specify structural restrictions on the instances of 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. A first feature that is missing in the OCL is a way to specify arbitrary 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). These events 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, they should preferably be added during the behavioral semantic definition. By making events explicit, it is possible to specify behavioral constraints on them (and particularly the constraints under which the system is amenable to analysis). Such constraints have to hold all along the execution and we call them *behavioral invariants*.

Some of these behavioral invariants are applicable to every model of the domain and should be expressed at the metamodel level but there exists other application-specific behavioral constraints. Application-specific constraints can be related to performance issues (*e.g.,* deadline) or to specific synchronization mechanisms dictated by the application (consider vehicle indicators for instance). Both kinds of behavioral invariants must be taken into account for a total understanding of a model. We propose to extend the OCL language with explicit event specification and behavioral invariants.

Amongst the benefits expected from such an extension of the OCL language, 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, ease the communication among domain specialists.

The next section describes related works that were also concerned either by the extension of the OCL or by the explicit specification of the model behaviors, at the metamodel level.

3 Related works

Our approach aims at extending the OCL so that the specification of behavioral invariants can be possible. 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; has already done by other approaches [16,17,18]. Adding temporal logics in the OCL is a good way to take benefits from a very expressive and formal language. However, it gets low acceptance level 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. This paper highlights 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 by [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 ing 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 call. 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 specifically associated with a specific behavior or message call.

Cariou *et al.* [11] 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. There is consequently a need for methods that can process these data. They use other specific behaviors (specified by their pre/post conditions) to specify when the data for the system state must be handled. 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, few synchronization capabilities, no way to specify concurrency explicitly and no temporal operators.

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]) while 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 detailing any "behavior" of the CPU. The goal of behavioral invariants is to specify the acceptable (partial) orders between the occurrences of the events.

Finally, when speaking about making domain-specific executable models, we should mention KerMeta [9]. 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. Also KerMeta does not support timed invariants or concurrency (like periodic execution, parallel execution, deadline). However, the be accessible as first-class citizens. Also KerMeta does not support timed invariants or concurrency (like periodic execution, parallel execution, deadline). 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

4.1 Rationale and overview of the approach

We want to be able to specify behavioral constraints (also called behavioral invariants) on structural models and metamodels. To do so, we use the notion of event, an event being a totally ordered set of event occurrences. Constraints on events specify the acceptable partial orders of the event occurrences in the system. These events are linked directly to elements of the structural model and can be annotated with their intentional meaning (start, stop, send, receive, etc) if needed. For instance, we may specify that an event occurrence that represents the sending of a signal on a connector occurs before the event occurrence that represents the reception of the same signal: therefore we build a causality constraint between these two events. Additionally, if we want to specify such constraints on the component metamodel of figure 1, the sending event can be binded to a structural element like the output port and tagged with “send” whereas the reception event is binded to an input port and tagged as "receive". If the metamodel is more detailed, the events could be binded to methods rather than purely structural elements.

In our proposition, events and constraints are specified using logical and multiform time. Logical time has proved its benefits in several domains. It was first introduced by Lamport to represent the execution of distributed systems [22]. It has then been extended and used in distributed systems to check the communication and causality path correctness [23]. During the same period, logical time has also been intensively used in synchronous languages [24,25] 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 [26]. 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.

relative to the physical time) but also the relative ordering of event occurrences.

We propose to use a variant of CCSL (Clock Constraint Specification Language) to specify the behavioral invariants. CCSL, is a model-based declarative language used to specify relationships between *clocks* (*i.e.,* events). CCSL has a formal semantics [27] that defines valid executions according to a given CCSL specification or rejects the specification when there are some contradictions. Its declarative nature makes it suitable for an integration in the OCL. A CCSL specification is a set of clocks and clock constraints that formally defines a set of possible interactions. 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. These user-defined relations can be grouped in a library. All the behavioral invariants for a given domain can consequently be grouped together in a domain-specific interaction library. A domain-specific interaction library is a reusable entity that can be used by several invariants and applied on various metamodels or models.

To equip a model with a specific behavioral invariant, some steps are mandatory. First, we have to define the meaningful events with regards to the behavioral invariant we want to express. During this step, it is possible to choose if the event reflects an activation, a start, a stop, a sending, a production, etc. Then the events are explicitly associated with a model element. Third, the events are then constrained with behavioral invariants. The model is then explicitly equipped with a formal and executable behavior on its events.

CCSL has been used successfully in several domains ranging from the expression of the Synchronous Data-Flow (SDF) model of computation [28] to the specification of temporal constraints for EAST-ADL models [29] but also to reflect the behavioral impact of implementation on requirements [30] and so on [31,32,33,34,35,36,37,38]. Based on these experiments, we have found that constraints from the domain should be expressed on the metamodel rather than on the model. For instance in the component model on Figure 2, we may want to equip all the connectors with the same behavioral invariants. This is the reason why we propose to integrate the CCSL constraints to the OCL language and then to enable the specification of constraints by intention on the metamodel rather than by extension on the model. On one hand it greatly simplifies the specification of the behavioral invariants for a specific domain, and on the other hand it makes a clean separation between the constraints implied by the domain and the ones induced by the application itself. the reader can notice that this separation is naturally done in OCL depending on whether constraints are specified at the metamodel or at the model level.

The simulation of such a CCSL model provides a possible interaction between the events of the model elements with respect to the behavioral invariant(s). This simulation is done at the model level by the associated framework, named TIMESQUARE. The result is a partially ordered trace directly linked to the specification. This link provides an interesting interactive feedback such as the drawing of timing or sequence diagrams, the animation of existing diagrams, or the possibility to execute user-defined code. TIMESQUARE also provides a mechanism to define new ways to interpret the results of the simulation, *i.e.,* to build user-defined back-ends.

After this brief overview of the proposed approach, the following sections elaborate

After this brief overview of the proposed approach, the following sections elaborate on different aspects of our approach. The explanation starts with an informal description of CCSL before introducing its underlying mathematical model and its integration within the OCL.

4.2 CCSL concepts and relations

In this paper, we consider only a small subset of CCSL, focusing on syntactic and semantic aspects sufficient to understand the examples given along the paper. The syntax and the semantics of full CCSL is available in a technical report [27] and an overview is available at `http://timesquare.inria.fr`.

A simplified view of the CCSL metamodel is given in Figure 3[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 (denoted $\equiv$). By combining such relationships, a time system is the specification of partially ordered sets of instants [39]. 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

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

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*.

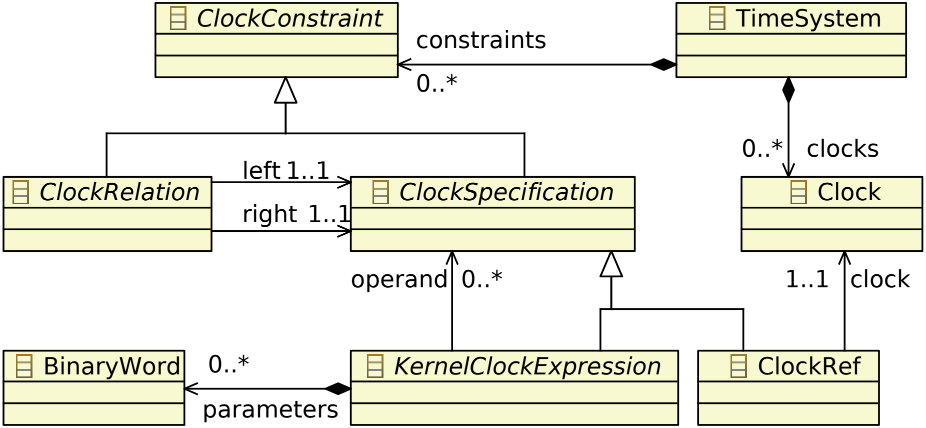


Fig. 3. Simplified CCSL metamodel

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

– the precedence (denoted $\boxed{\prec}$), which specifies that all instants of the left clock occur before the corresponding instants of the right clock:
  $c1 \boxed{\prec} c2 \Rightarrow \forall k \in \mathbb{N} \setminus 0, c1[k] \prec c2[k]$
– the coincidence (denoted $\boxed{=}$), which specifies that all instants of the left clock are simultaneous with the corresponding instants of the right clock:
  $c1 \boxed{=} c2 \Rightarrow \forall k \in \mathbb{N} \setminus 0, c1[k] \equiv c2[k]$

We also only use two clock expressions:

– filteredBy expression (denoted $ref \blacktriangledown bw$), which takes a clock reference (ref) as an operand and a binary word (bw) as a parameter. The resulting clock ticks synchronously with the clock operand pruned from some instants. Which instants are kept and which are pruned depend on the binary word parameter.
– delayedFor expression (denoted $c \ \$ \ n \ \mathit{on} \ ref$), which takes two clocks (c and ref) as operands and an integer (n) as 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 CCSL specification $\mathcal{S}$ below.

$$\mathcal{S} = \left(A \boxed{\prec} B \blacktriangledown (10)^{\omega}\right) \qquad (1)$$

$(10)^{\omega}$ denotes the infinite (periodic) binary word $10101 \cdots 101 \cdots$. 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.

$$\mathcal{S} = \left(C \boxed{=} B \blacktriangledown (10)^{\omega}\right) \mid A \boxed{\prec} C \tag{2}$$

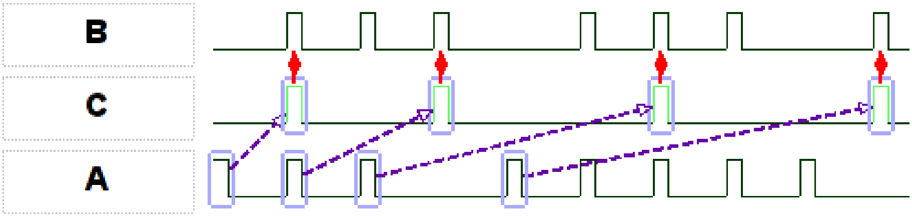


Fig. 4. A timing diagram of the example in equation 2

Equation 2 is very flexible since, as represented in Figure 4, the clock A can have any advance on the other clocks. Of course, if needed, this advance could be bounded.

It is clear that these low abstraction level constraints are not intended to be used by non specialists. This is the goal of the libraries to provide domain-specific constraints, constructed by semantic specialists but used by (meta)model designers.

4.3 CCSL operational semantics

This paragraph gives the flavor of the CCSL semantics without falling into the details. Please refer to [27] 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.

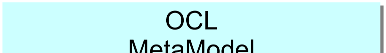
To be as close as possible to the mathematical rules while staying at the model level, we added the implementation code of the semantics as encapsulation of the object from the EMF model (Eclipse Modeling Framework). It allows a CCSL specification to be executed in the same technological space than other models and, for each reaction step, the result (itself provided as a trace model [34]) is a set of ticking clocks as well as internal variables that can be accessed to provide feedback to users (cf. section 4.6).

4.4 Integration of CCSL in OCL

Our goal is to integrate CCSL in the OCL as naturally as possible to ease its acceptance. A first solution consists in creating a specific OCL library that extends the existing OCL standard library with the notion of event and some possible constraints on them.

However, this method does not allow the use of domain-specific event constraints (*i.e.,* like the constraints of a CCSL library). To avoid repulsing users with difficult specifications, and allowing the use of domain-specific constraints, we extended the OCL metamodel itself. Instead of creating a new invariant named behavioral invariant, we chose to behavioral constraint as a specialization of the existing invariants. More precisely we chose to extend the OCL concept of expression (*ExpCS*) with the *Constraint* concept from the CCSL metamodel. Additionally, we added Event as a primitive type so that it can be declared naturally (see section 4.5 for an example). Of course to make it usable, we also modified the concrete syntax definition. All these changes have been prototyped in the current OCL implementation of the eclipse foundation[2].

The result of the metamodel integration is the possibility to specify behavioral invariants at the metamodel level, in an OCL-like fashion. Of course, such invariants cannot be directly simulated by TIMESQUARE since the expressed constraints represents the semantics of the models that conform the metamodel. To reuse the existing CCSL tooling (*i.e.,* TIMESQUARE)), we propose to realize a high order transformation whose inputs are the OCL constraints and the domain-specific metamodel and the result is a transformation from a domain-specific model to a CCSL specification (see Fig. 5). Because OCL and the QVT transformation language [40] share a same query language, operational QVT is a natural candidate to specify the transformations.



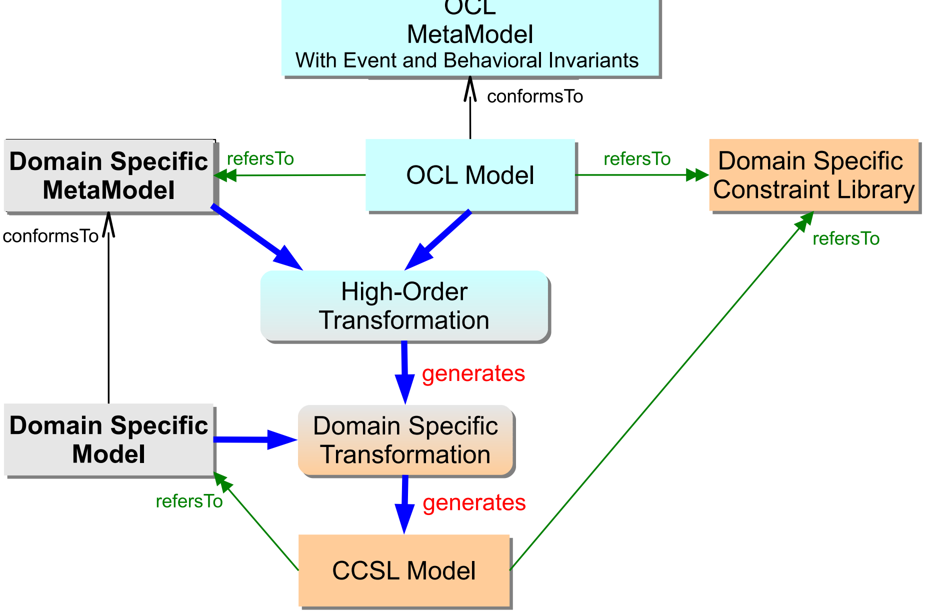

OCL
MetaModel
With Event and Behavioral Invariants
conformsTo
Domain Specific MetaModel
refersTo
OCL Model
refersTo
Domain Specific Constraint Library
conformsTo
refersTo
High-Order Transformation
generates
Domain Specific Model
Domain Specific Transformation
refersTo
generates
CCSL Model

Fig. 5. models and transformations to take benefits from CCSL tooling

[2] `http://www.eclipse.org/modeling/mdt/?project=ocl`

4.5 Applying behavioral invariants at the metamodel level

This section explains how a specific model, like the one introduced in Figure 2, can be equipped with the behavioral invariants defined in a domain-specific interaction library. Because this first example is simplistic, we used the CCSL constraints from the kernel library presented in section 4.2. In this section, we use the textual syntax of CCSL implemented with XText and not the mathematical syntax introduced in Section 4.2.

To specify a behavioral invariant, we use three constructs from the OCL language. The first one is the *definition expression*. It can be used to add a new property to the element of its context (keyword *def*). It is used to specify the events (*Event*) later used by behavioral invariants. An event is initialized by a CCSL expression, which associates with it a specific model element (*self* in line 3 and 4 of the listing example 1.1). The initialization can also enrich the event with its intentional meaning with regards to the model element (*start* on line 3 and *stop* on line 4). The second one is the use of the *let* expression, which allows to stock a variable initialized by an expression. We use it either to ease the reading of the constraint by renaming events retrieved by model navigation (or query) or to create an intermediate event. Creation of intermediate events is allowed because in OCL variables can be initialized by an expression, and the CCSL expression (*Expression*) concept extends the OCL expressions. Of course, the third OCL concept used is the invariant (*inv*) to specify constraints (*Relation*) between events.

The first behavioral invariants we want to specify states that all the starting events of each sub-component of a given component are synchronous (*i.e.,* all the sub-components are activated simultaneously). The second invariant states that the execution duration is each sub-component of a given component are synchronous (*i.e.,* all the sub-components are activated simultaneously). The second invariant states that the execution duration is not instantaneous but is bounded by a constant, provided as an integer property named *worstCaseExecutionTime* in the metamodel. Finally, each component has a *local* clock on which the execution time is counted. To do so, we use the kernel relations *coincides* and *precedes* as well as the *delayedFor* expression.

The first step consists in choosing the model elements that are relevant from a behavioral constraints point of view. In this example, only the components are relevant so that in the following pseudo code only the Component context is used. During the second step, for each of the components, we create two events, which reference the corresponding model element (*i.e.,* the component itself). The first event reflects the *start* of the execution and the second one its end (*i.e., finish*). Finally, we constrained the events to specify the expected interactions between these events. The first invariant (*componentSynchronization*) specify the synchronous start of all the sub-components of a hierarchical component while the second one (*executionDuration*) constrains the execution duration of a component to be less than the one specified in the component *worstCaseExecutionTime* property.

Listing 1.1. Example of behavioral invariants integrated in OCL

```
context Component
 def: e_start : Event = new Event(self):start
 def: e_stop :  Event =  new Event(self):finish
 def: e_localClock : Event = new Event(self)

 inv componentSynchronization:
    self.internalComponents->forall( c1, c2 |
        c1 <> c2 implies (
            Relation r1[coincides](c1.e_start, c2.e_start)
            )
        )

  inv executionDuration:
    let wcet:Integer = self.worstCaseExecutionTime in
    let begin:Event =  self.e_start in
    let end:Event =  self.e_stop in
    let ref:Event = self.e_localClock in
    let beginDelayed:Event =
      Expression e1[DelayedFor](begin,wcet,ref) in
     Relation r2[precedes](end, beginDelayed)
```

Listing 1.1 uses only constraints from the CCSL kernel. To simplify the writing and the reading of the specification, the expression $e1$ and the relation $r2$ could be encapsulated in a domain-specific relation. If we name it $BoundExecTime$, the specification of the execution invariant becomes the one in the listing 1.2. It shows the kind of gain obtained by the use of domain-specific constraints, which can be arbitrarily complex without adding complexity in their use.

Listing 1.2. Example of behavioral invariants integrated in OCL

Listing 1.2. Example of behavioral invariants integrated in OCL

```
    inv executionDuration:
     let wcet:Integer = self.worstCaseExecutionTime in
     let begin:Event =  self.e_start in
     let end:Event =  self.e_stop in
     let ref:Event = self.e_localClock in
      Relation r2[BoundExecTime](begin,end,wcet,ref)
```

Let us note that, in this example, every component has a local clock on which the execution time is counted. This can of course be latter constrained, for instance to reflect that all *localClock*s are synchronized. It could also be constrained to reflect the allocation of the component on a single processor so that a local clock ticks only when the scheduler actually executes the component (see [32] for details).

For the sake of simplicity, the previous specification does not remove all the model ambiguities. However, the behavioral invariants make explicit the possible interactions of the events in the systems for all component models that conforms to the metamodel. These constraints can then be used:

– as a formal reference linked to the model for system designers to handle the behavior of the system;
– as a formal reference when transformations must be performed to other analysis-specific formal languages;
– as a means to simulate the component model and then to provide feedback to the user; useful for debugging its model.

In the tool presented in the next section, the creation of a CCSL specification from the OCL behavioral invariants is manual. However, the automation of this task (*i.e.,* the implementation of the high order transformation) is currently under progress and is expected to be finished in the next months.

4.6 Existing tooling and facilities

TimeSquare is the software environment we have developed to support the modeling approach presented in the previous subsections. This tool is specifically developed to formally handle models so that the output is also a model, linked to the CCSL specification model. It allows the simulation to remain in the same technological space and greatly eases the feedback to users. TimeSquare has four main features:

1. definition/modeling of user-defined libraries,
2. specification/modeling of a CCSL model and its application to a specific structural (meta)model,
3. simulation of CCSL models and generation of the corresponding execution trace model,
4. based on the trace model, display and exploration of timing and sequence diagrams, animation of UML-based models, enhancement of the original UML model based on analysis results, and execution of user-defined code.

If an actual execution is instrumented to produce a trace in the OTF format [41], If an actual execution is instrumented to produce a trace in the OTF format [41], TIMESQUARE can also be used to check the conformance of the trace with respect to a behavioral specification. All the feedback is based on a back-end manager and the tool supports the definition of new user-defined back-ends as plugins. TIMESQUARE is provided as a set of Eclipse plugins. A detailed description of TIMESQUARE features, download site, examples, and video demonstrations are available on its website:

`http://timesquare.inria.fr/`

5 Conclusion

This paper presents a model-driven approach that defines two extensions of the OCL language. The first one is used to explicitly specify the events of a model and the second one is used to specify behavioral invariants. Behavioral invariants is a way to specify constraints on the partial ordering of the event occurrences in the system. The goal is to specify the expected behavior of a model, so that domain specific analysis can be conducted. Such behavior can then be used in various scenari ranging from model animation to generation of observers at runtime. The extension is based on CCSL (the *Clock Constraint Specification Language*), 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 used to specify (not program) the expected behavior of the whole model. This extension aims at complementing (not replacing) other features of the OCL language like the use of pre/post conditions.

TIMESQUARE, which is available for download, is the effective Model Development Kit based on eclipse that supports the approach. It allows the specification of CCSL specifications and libraries and provides convenient feedback to the users about his/her modeling (either during the modeling of a system with a specific semantics or during the modeling of a new behavioral semantics). The final phase of the integration of the OCL extensions is currently under implementation in TIMESQUARE and a prototype is expected before the september OMG meeting in order to make a demonstration.

There are two main ongoing investigations to extend this work. As already stated, a short-term objective is to finish the automation of the transformation from the extended OCL to a CCSL specification for a given model. The direct results of this short-term objective is a study about using different domain specific interaction libraries in a single model (*i.e.,* heterogeneous modeling). Another futur work consists in studying extensions of the CCSL language like the capacity to deal with continuous time solver or to use the notion of mode in order 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

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. OMG, Unified Modeling Language, Superstructure, Version 2.1.2 formal/2007-11-02 (November 2007).
9. P.-A. Muller, F. Fleurey, J.-M. Jézéquel, Weaving executability into object-oriented metalanguages, in: L. C. Briand, C. Williams (Eds.), MoDELS, Vol. 3713 of Lecture Notes in Computer Science, Springer, 2005, pp. 264–278.
10. OMG, Object Constraint Language, Version 2.3.1 formal/2012-01-01 (January 2012).
11. 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.
12. 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.
13. 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.

14. T. Grötker, System design with SystemC, Kluwer Academic Publishers, 2002.
15. A. Jantsch, Modeling embedded systems and SoC's: concurrency and time in models of computation, Morgan Kaufmann (an imprint of elsevier science), 2004.
16. M. Cengarle, A. Knapp, Towards ocl/rt, in: L.-H. Eriksson, P. Lindsay (Eds.), FME 2002:Formal MethodsGetting 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. L. Lamport, Time, clocks, and the ordering of events in a distributed system, Communications of the ACM 21 (7) (1978) 558–565.
23. C. Fidge, Logical time in distributed computing systems, Computer 24 (8) (2002) 28–33.
24. 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)

24. 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.
25. G. Berry, The foundations of Esterel, Proof, Language and Interaction: Essays in Honour of Robin Milner (2000) 425–454.
26. F. Boussinot, R. De Simone, The ESTEREL language, Proceedings of the IEEE 79 (9) (2002) 1293–1304.
27. C. André, Syntax and Semantics of the Clock Constraint Specification Language (CCSL), Research report, INRIA and University of Nice (May 2009).
28. 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`
29. F. Mallet, M.-A. Peraldi-Frati, C. Andr, 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.
30. 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`
31. 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`
32. 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`

33. 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`
34. 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`
35. 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`
36. 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 Sci-

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`

37. 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`
38. H. Yu, J.-P. Talpin, L. Besnard, T. Gautier, F. Mallet, C. Andre 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`.
39. 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.
40. OMG, Meta Object Facility (MOF) 2.0 Query/View/Transformation, v1.1 , Version 1.1 formal/2011-01-01 (January 2011).
41. A. Knupfer, R. Brendel, H. Brunst, H. Mix, W. Nagel, Introducing the Open Trace Format (OTF), in: V. Alexandrov, G. van Albada, P. Sloot, J. Dongarra (Eds.), Computational Science (ICCS), Vol. 3992 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2006, pp. 526–533.
URL `http://dx.doi.org/10.1007/11758525_71`