The Time Model of Logical Clocks available in the OMG MARTE profile

Charles André, Julien Deantoni, Frédéric Mallet, Robert de Simone

The Time Model of Logical Clocks available in the OMG MARTE profile

Charles André[1,2], Julien DeAntoni[1,2], Frédéric Mallet[1,2], and Robert de Simone[1]

[1] INRIA Sophia Antipolis Méditerranée, F-06902 Sophia Antipolis, France
firstname.surname@sophia.inria.fr,
home page: http://www.inria.fr/sophia/aoste/
[2] Université de Nice-Sophia Antipolis, Laboratoire I3S, UMR 6070 CNRS, F-06903 Sophia Antipolis Cedex, France

Summary. Multiform logical time, introduced and made popular through its central role in Synchronous Language theory, is already present in many formalisms pertaining to embedded system design, although usually in a hidden fashion. Logical time considers time bases that can be generated from any sort of sequences of events, not necessarily equally spaced in physical time. Our main goal here is to capture some of the essence of multiform logical time, and encapsulate it into a dedicated syntax (CCSL, Clock Constraint Specification Language, part of the UML profile for MARTE). CCSL provides ways to express loose or strict constraints between distinct logical clocks. Solving such clock constraints amounts to relating clocks to a common reference one, which then can be thought of as closer to physical. We motivate the role of MARTE Time Model and CCSL by using them to explain and formally characterize important semantic features of East-ADL/AUTOSAR, AADL, and Ptolemy's SDF models.

The full paper is available on the site of the publisher :
https://doi.org/10.1007/978-1-4419-6400-7_7

1 Introduction

Embedded System Design is progressively becoming a field of choice for Model-Driven Engineering techniques. There are fundamental reasons for this trend:

1. Target execution platforms in the embedded world are often heterogeneous, flexible or reconfigurable by nature (as opposed to the conventional Von Neumann computer architecture). Such architectures can even sometimes be decided upon and customized to the specific kind of applications meant to run on them. Early architecture modeling allows exploring possible variations, possibly optimizing the form of future execution platforms

before they are actually built. Architecture exploration is becoming a key ingredient of embedded system design, where applications and execution platforms are essentially designed jointly and concurrently at model level;
2. Applications are often reactive by nature, that is, meant to react repeatedly with an external environment. The main design concern goes with handling data or control flow propagation, which includes frequently streaming and pipelined processings. In contrast, the actual data values and computation contents are of lesser importance (for design, not for correctness). Application models such as process networks, reactive components and state/activity diagrams are used to represent the structure and the behavior of such applications (more than usual software engineering notions such as classes, objects, and methods);
3. Designs are usually subject to stringent real-time requirements, imposed "from above", while they are constrained by the limitations of their components and the availability of execution platform resources, "from below". Allocation models can here serve to check at early stages whether there exist feasible mappings and schedulings of application functions to architecture resources and services that may match the requirements under the given constraints.

The model-driven approach to embedded system engineering is often tagged as Y-Chart methodology, or also Platform-Based design. In this approach, application and architecture (execution platform) models are developed and refined concurrently, and then associated by allocation relationships, again at a virtual modeling level.

The representation of requirements and constraints in this context becomes itself an important issue, as they guide the search for optimal solutions inside the range of possible allocations. They may be of various natures, functional or extra-functional. In this article, we focus on requirements and constraints which we call logical functional timing, and which we feel to be an important (although too often neglected or misconceived) aspect of embedded system modeling.

Timing information in modeling is often used as extra-functional, real-time annotations analyzed mostly by simulation. But the relevance of these timing figures in later implementations has to be further assessed then. On the other hand, time information also carries functional intent, as it selects some behaviors and discards others. Very often this can be done not only by using single form physical time, but also logical multiform time models. For instance, it may be sufficient to state that a process runs twice as fast as another, or at least as fast as the second, without providing concrete physical time figures. The selected solutions will work for any such physical assignment that matches the logical time constraints. Also, durations may be counted with events that are not regular in physical time: the number of clock cycles of a processor in a low-power design context may vary according to frequency scaling or clock gating; processing functions may be indexed by the engine

crankshaft revolution angle in case of automotive applications. Other examples abound in the embedded design world. Modeling with logical time partial ordering was advocated in [16]. The notion of multiform (or polychronous) logical time has been exploited extensively in the theory of Synchronous languages [2], in HDLs (Hardware Description Languages), but also importantly in the many approaches of model-based scheduling theories around process networks [6, 26] and formal data/control-flow graphs synthesized from nested loop programs [8]. Software pipelining and modulo scheduling techniques are based on such logical time counted in parallel processor execution cycles. The important common feature of all these approaches is that (logical) time is an integral part of the functional design, to be used and maintained along compilation/synthesis, and not only simulation. In many cases the designer is fronted with time decisions in order to specify correct behaviors. This is of course largely true also of classical physical real-time requirements, but their operational demand is usually downplayed as they are only considered for analysis, not synthesis. Many of the techniques are still shared between both worlds (for instance consider zero-time abstraction of atomic behaviors, and progress all behaviors in correct causal order in a run-to-completion mode before time may pass in a discrete step). Some techniques are still different, and certainly the goals are, but the underlying models are similar. Large and heterogeneous systems require a single common environment to integrate all these models while still preserving the semantics and the analysis techniques of each of them.

We consider here the use of existing modeling tools to integrate all these models. The UML appears as a good candidate since it has unified in a common syntactic notation most of the underlying formal models generally used for embedded systems: state machines, data-flow graphs (UML activities), static models to describe the execution platforms (block diagrams). It is even more relevant because the UML profile for MARTE, recently adopted by the OMG, proposes extensions to UML specifically targeting real-time and embedded systems. Our goal is to provide an explicit formal semantics to the UML elements so that it can be referred to as a golden model by external tools and make the model amenable to formal analysis, code generation or synthesis. An explicit and formal semantics within the model is fundamental to prevent different analysis tools from giving a different semantics to the same model. Each analysis technique relies on a specific formal model that has its own model of computation and communication (MoCC). We propose a language, called Clock Constraint Specification Language (CCSL), specifically devised to equip a given model (conformant to the UML or to any domain-specific language) with a timed causality model and thus define a specific MoCC. When considering UML models, CCSL relies on the MARTE time subprofile to identify model elements on which CCSL constraints apply.

In this chapter, we select different models from different domains to illustrate possible uses of CCSL. First, we show how it can be used to express classical constraints of the real-time and embedded domain by expressing East-

ADL [27] extra-functional properties in CCSL. East-ADL proposes a set of time requirements classical in automotive applications (duration, deadline, jitter) and on which time requirements for AUTOSAR[3] are being built. The second illustration falls in the avionics domain and focuses on AADL (Architecture & Analysis Description Language) [9]. AADL is an Architecture Description Language (ADL) adopted by the SAE (Society of Automotive Engineering) that offers specific support for schedulability analysis. It also considers classical computation (periodic, sporadic, aperiodic) and communication (immediate/delayed, event-based or timed-triggered) patterns. However, it departs from East-ADL because it explicitly considers the execution platform to which the application is allocated. Our illustration uses MARTE (and notably its allocation subprofile) to build a model amenable to architecture exploration and schedulability analysis. These first two examples consider models that combine logical and physical time. The last example considers a purely logical case. A CCSL library that specifies the operational semantics of the Synchronous Data Flow (SDF [15]) is built. This library is applied to several diagrammatic views, equipping purely syntactic models with an explicit behavioral semantics.

Section 2 gives a general overview of MARTE, a detailed view of the time subprofile and of its facilities to annotate UML models with (logical) time. Then, Section 3 describes the CCSL syntax and semantics together with the mechanisms for building libraries. It also introduces TimeSquare, the dedicated environment we have built to analyze and verify UML/MARTE/CCSL models. The following sections address several possible usages of CCSL and describe CCSL libraries for the different sub-domains: automotive and East-ADL in Section 4, avionics and AADL in Section 5, static analysis and SDF in Section 6.

2 The UML Profile for MARTE

2.1 Overview

The Unified Modeling Language (UML) [23] is a general-purpose modeling language specified by the Object Management Group (OMG). It proposes graphical notations to represent all aspects of a system from the early requirements to the deployment of software components, including design and analysis phases, structural and behavioral aspects. As a general-purpose language, it does not focus on a specific domain and maintains a weak, informal semantics to widen its application field. However, when targeting a specific application domain and especially when building trustworthy software components or for critical systems where life may be at stake, it is absolutely required to extend the UML and attach a formal semantics to its model elements. The simplest and most efficient extension mechanism provided by the UML is through the definition of profiles. A UML profile adapts the UML to a specific domain by

[3] `http://www.autosar.org`

adding new concepts, modifying existing ones and defining a new visual representation for others. Each modification is done through the definition of annotations (called stereotypes) that introduce domain-specific terminology and provide additional semantics. However, the semantics of stereotypes must be compatible with the original semantics (if any) of the modified or extended concepts.

The UML profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE [22]) extends the UML with concepts related to the domain of real-time and embedded systems. It supersedes the UML profile for Schedulability, Performance and Time (SPT [20]) that was extending the UML 1.x and that had limited capabilities.

MARTE has three parts: *Foundations*, *Design* and *Analysis*. The foundation part is itself divided into five chapters: *CoreElements*, *NFP*, *Time*, *Generic Resource Modeling* and *Allocation*. *CoreElements* defines configurations and modes, which are key parameters for analysis. In real-time systems, preserving the non-functional (or extra-functional) properties (power consumption, area, financial cost, time budget...) is often as important as preserving the functional ones. The UML proposes no mechanism at all to deal with non-functional properties and relies on mere string for that purpose. NFP (Non Functional Properties) offers mechanisms to describe the quantitative as well as the qualitative aspects of properties and to attach a unit and a dimension to quantities. It defines a set of predefined quantities, units and dimensions and supports customization. NFP comes with a companion language called VSL (Value Specification Language) that defines the concrete syntax to be used in expressions of non-functional properties. VSL also recommends syntax for user-defined properties. Time is often considered as an extra-functional property that comes as a mere annotation after the design. These annotations are fed into analysis tools that check the conformity without any actual impact on the functional model: *e.g.,* whether a deadline is met, whether the end-to-end latency is within the expected range. Sometimes though, time can also be of a functional nature and has a direct impact on what is done and not only when it is done. All these aspects are addressed in the time chapter of MARTE. The next section elaborates on the time profile.

The design part has four chapters: High Level application modeling, Generic component modeling, Software Resource Modeling, and Hardware Resource Modeling. The first chapter describes real-time units and active objects. Active objects depart from passive ones by their ability to send spontaneous messages or signals, and react to event occurrences. Normal objects, the passive ones, can only answer to the messages they receive. The three other parts provide a support to describe resources used and in particular execution platforms on which applications may run. A generic description of resources is provided, including stereotypes to describe communication media, storages and computing resources. Then this generic model is refined to describe software and hardware resources along with their non-functional properties.

The analysis part also has a chapter that defines generic elements to perform model-driven analysis on real-time and embedded systems. This generic chapter is specialized to address schedulability analysis and performance analysis. The chapter on schedulability analysis is not specific to a given technique and addresses various formalisms like the classic and generalized Rate Monotonic Analysis (RMA), holistic techniques, or extended timed automata. This chapter provides all the keywords usually required for such analyses. In Section 5, we follow a rather different approach and instead of focusing on syntactic elements usually required to perform schedulability analysis (periodicity, task, scheduler, deadline, latency), we show how we can use MARTE time model and its companion language CCSL to build libraries of constraints that reflect the exact same concepts. Finally, the chapter on performance analysis, even if somewhat independent of a specific analysis technique, emphasizes on concepts supported by the queueing theory and its extensions.

MARTE extends the UML for real-time and embedded systems but should be refined by more specific profiles to address specific domains (avionics, automotive, silicon) or specific analysis techniques (simulation, schedulability, static analysis). The three examples addressed here consider different domains and/or different analysis techniques to motivate the demand for a fairly general time model that has justified the creation of MARTE time subprofile.

2.2 The Time profile

Time in SPT is a *metric* time with implicit reference to physical time. As a successor of SPT, MARTE supports this model of time. UML 2, issued after SPT, has introduced a model of time called *SimpleTime* [23, Chap. 13]. This model also makes implicit reference to physical time, but is too simple for use in real-time applications, and was initially devised to be extended in dedicated profiles.

MARTE goes beyond SPT and UML 2. It adopts a more general time model suitable for system design. In MARTE, Time can be *physical*, and considered as *continuous* or *discretized*, but it can also be *logical*, and related to user-defined clocks. Time may even be *multiform*, allowing different times to progress in a non-uniform fashion, and possibly independently to any (direct) reference to physical time.

In MARTE, time is represented by a collection of *Clocks*. Each clock specifies a totally ordered set of instants. There may be dependence relationships between instants of different clocks. Thus this model, called the MARTE time structure, is akin to the Tagged Systems [16]. To cover continuous and discrete times, the set of instants associated with a clock can either be dense or discrete. In this paper, most clocks are discrete (*i.e.,* they represent discrete time). In this case the set of instants is indexed by natural numbers. For a clock c, $c[k]$ denotes its k^{th} instant.

The MARTE Time profile defines two stereotypes ClockType and Clock to represent the concept of clock. ClockType gathers common features shared by a

family of clocks. The ClockType fixes the nature of time (dense or discrete), says whether the represented time is linked to physical time or not (respectively identified as chronometric clocks and logical clocks), chooses the type of the time units. A Clock, whose type must be a ClockType, carries more specific information such as its actual unit, and values of quantitative (resolution, offset, etc.) or qualitative (time standard) properties, if relevant.

TimedElement is another stereotype introduced in MARTE. A timed element is *explicitly bound* to at least one clock, and thus closely related to the time model. For instance, a TimedEvent, which is a specialization of TimedElement extending UML Event, has a special semantics compared to usual events: it can occur only at instants of the associated clock. In a similar way, a TimedValueSpecification, which extends UML ValueSpecification, is the specification of a set of *time values* with explicit references to a clock, and taking the clock unit as time unit. Thus, in a MARTE model of a system, the stereotype TimedElement or one of its specializations is applied to model elements which have an influence on the specification of the temporal behavior of this system.

The MARTE Time subprofile also provides a model library named TimeLibrary. This model library defines the enumeration TimeUnitKind which is the standard type of time units for chronometric clocks. This enumeration contains units like s (second), its submultiples, and other related units (minute, hour...). The library also predefines a clock type (IdealClock) and a clock (idealClk) whose type is IdealClock. idealClk is a dense chronometric clock with the second as time unit. This clock is assumed to be an ideal clock, perfectly reflecting the evolutions of physical time. idealClk should be imported in user's models with references to physical time concepts (*i.e.,* frequency, physical duration, etc.). This is illustrated in Sections 4 and 5.

3 CCSL Time Model

3.1 The Clock Constraint Specification Language

CCSL is a language to impose dependence relationships between instants of different clocks. This dependency is specified by *Clock constraints*. A ClockConstraint is a TimeElement that extends UML Constraint. The constrained elements are clocks. A clock constraint imposes relationships between instants of its constrained clocks. So, to understand clock constraints, we have first to define relations on instants.

Instant Relations

The *precedence* relation $\preccurlyeq$ is a reflexive and transitive binary relation on set of instants. From $\preccurlyeq$ we derive four new relations: *Coincidence* ($\equiv \triangleq \preccurlyeq \cap \succcurlyeq$), *Strict precedence* ($\prec \triangleq \preccurlyeq \setminus \equiv$), *Independence* ($\| \triangleq \overline{\preccurlyeq \cup \succcurlyeq}$), and *Exclusion* ($\# \triangleq \prec \cup \succ$). The precedence relation represents causal dependency. The

coincidence relation is a strong relation that forces simultaneous occurrences of instants from different clocks.

Clock relations

Specifying a full time structure using only instant relations is not realistic. Moreover a set of instants is usually infinite, thus forbidding an enumerative specification of instant relations. Hence the idea to extend relations to clocks. CCSL defines five basic clock relations. In the following definitions, a and b stand for Clocks. For simplicity, mathematical expressions are given only in the case of discrete clocks.

- *Subclocking:* $a \boxed{\subset} b$ means that each instant of a is coincident with an instant of b, and that the coincidence mapping is order-preserving. a is said to be a subclock of b, and b a superclock of a.
- *Equality:* $a \boxed{=} b$ is a special case of subclocking where the coincidence mapping is a bijection. $\forall k \in \mathbb{N}^\star, a[k] \equiv b[k]$. a and b are "synchronous".
- *Precedence:* $a \boxed{\preccurlyeq} b$ means $\forall k \in \mathbb{N}^\star, a[k] \preccurlyeq b[k]$. a is said to be faster than b.
- *Strict precedence:* $a \boxed{\prec} b$ is similar to the previous one but considers the strict precedence instead. $\forall k \in \mathbb{N}^\star, a[k] \prec b[k]$.
- *Exclusion:* $a \boxed{\#} b$ means that a and b have no coincident instants.

The *Alternation* $a \boxed{\sim} b$, used in the application sections, is a derived clock relation that imposes $\forall k \in \mathbb{N}^\star, a[k] \prec b[k] \prec a[k+1]$. a alternates with b.

Clock expressions

They allow definitions of new clocks from existing ones. For instance, *Filtering* is an often used clock expression. Let a be a clock, and w a binary word (*i.e.,* a finite or infinite word on bits: $w \in \{0,1\}^* \cup \{0,1\}^\omega$). w is used as a *filtering pattern*. $a \blacktriangledown w$ defines a new clock, say b, such that $\forall k \in \mathbb{N}^\star, b[k] \equiv a[w \uparrow k]$, where $w \uparrow k$ is the index of the k^{th} 1 in w. Binary words are convenient representations of Boolean flows and schedules. A schedule is an activation sequence, generally periodic in which case *periodic* binary words are used, denoted as $w = u(v)^\omega$, where u (prefix) and v (period) are finite binary words. w is the infinite binary word $u \bullet v \bullet \cdots \bullet v \bullet \cdots$. Periodic binary words have already been successfully applied to N-Synchronous Kahn networks [3].

Clock constraints

A CCSL specification consists of a set of Clocks and a conjunction of clock constraints. A clock constraint is a clock relation between two clocks or clock expressions. The stereotype ClockConstraint has Boolean meta-attributes

(isCoincidenceBased and isPrecedenceBased) that indicate the kind of constraint. The coincidence-based constraints are also known as "synchronous" constraints, whereas the precedence-based constraints are called "asynchronous". There also exist mixed constraints in which case the two meta-attributes are set to true. A third meta-attribute (isChronometricBased) is used only for chronometric clocks and quantitative time constraints such as stability, skew.

Temporal evolutions

A CCSL specification imposes a complex ordering on instants. We do not explicitly represent this time structure. We compute possible *runs* instead. A run is a sequence of steps. Each step is a set of clocks that are simultaneously fired without violating any clock constraint. When a discrete clock ticks (or fires), the index of its current instant is incremented by 1. The computation of a step is detailed in a technical report [1] that provides a syntax and an operational semantics for a kernel of CCSL. Here, we just sketch this process.

Using the semantics of CCSL, from a clock constraint specification $\mathcal{S}$ we derive a logical representation of the constraints $[\![\mathcal{S}]\!]$. This representation is a Boolean expression on a set of Boolean variables $\mathcal{V}$, in bijection b with the set of clocks $\mathcal{C}$. Any valuation $v : \mathcal{V} \to \{0,1\}$ such that $[\![\mathcal{S}]\!](v) = 1$ indirectly represents a set of clocks F that respects all the clock constraints: $F = \{c \in \mathcal{C} \mid v(b(c)) = 1\}$. F is a possible set of simultaneously fireable clocks. Most of the time, this solution is not unique. Our solver supports several policies for choosing one solution.

CCSL libraries

CCSL specifications are executable specifications. However, the expressiveness of the kernel CCSL is limited, for instance by the lack of support for parameterized constructs. The full CCSL overcomes these limitations through libraries. A library is a collection of parameterized constraints, using constraints from one or many other libraries. The primitive constraints, which constitute the kernel CCSL, are grouped together in the *kernel library*. The operational semantics is explicitly defined only for the constraints of the kernel library. Each user-defined constraint is structurally expanded into kernel constraints, thus defining its operational semantics.

As a very simple example, we define a ternary coincidence relation, denoted $\boxed{\equiv}$. The user-defined relation has three clock parameters (v1, v2 and v3). This definition contains two instances of the equality relation whose definition is given in the kernel library. The textual specification of the ternary coincidence relation as follows:

$$\mathsf{def}\ \boxed{\equiv}\ (\mathsf{clock}\ v1,\ \mathsf{clock}\ v2,\ \mathsf{clock}\ v3)\ \triangleq\ (v1\ \boxed{=}\ v2)\ |\ (v1\ \boxed{=}\ v3)$$

CCSL libraries are used in the remainder of the paper either to group domain specific constraints (Section 4) or to encapsulate a specific MoCC (Section 6).

3.2 TimeSquare

TimeSquare is a software environment dedicated to the resolution of CCSL constraints and computation of partial solutions. It has four main features: 1) definition/modeling of CCSL user-defined libraries that encapsulate the MoCCs, 2) specification/modeling of a CCSL model and its application to a specific UML-based or DSL model, 3) simulation of MoCCs and generation of a corresponding trace model, 4) based on a trace model, displaying and exploring the augmented timing diagram, animating UML-based model and storing the scheduling result in the model and sequence diagrams.

TimeSquare is released as a set of Eclipse plug-ins. A detailed description of TimeSquare features, examples, and video demonstrations are available from its website (`http://www-sop.inria.fr/aoste/dev/time_square`).

Summary

In MARTE, clock constraints impose physical time or logical time (causal) relationships between timed elements. CCSL is a formal specification language, so that clock constraints expressed in CCSL are executable in the TimeSquare environment. They are also amenable to analysis. Note that while fully integrated with concepts from the MARTE profile, CCSL can be used outside the UML, for instance within the framework of a DSL. The purpose remains the same: to provide a time causality model fitting a model-driven approach.

4 MARTE and East-ADL2

We consider here an example from the automotive domain. We build a CCSL library for expressing the semantics of East-ADL time requirements. Their semantics is left informal in the East-ADL specification and we had to disambiguate some of their definitions to build our CCSL model. By building this library we make East-ADL requirement specifications executable and allow the use of TimeSquare to execute and animate UML models annotated with East-ADL stereotypes.

4.1 East-ADL2

East-ADL (Electronic Architecture and Software Tools, Architecture Description Language) has been initially developed in the context of the East-EEA European project [28]. To integrate proposals from the emerging standard AUTOSAR and from other requirement formalisms like SysML [29, 21], a new release called East-ADL2 [4, 27] has been proposed by the ATESST project. In this section, we abusively refer to both versions under the name East-ADL.

Structural modeling in East-ADL covers both analysis and design levels. In this chapter the focus is on the analysis level and especially on timing requirements.

Timing Requirements

East-ADL requirements extend SysML requirements and express conditions that must be met by the system. They usually enrich the functional architecture with extra-functional characteristics such as variability and temporal behavior. We focus on the three kinds of timing requirements available in East-ADL:

1. DelayRequirement that constrains the delay "from" a set of entities "until" another set of entities. It specifies the temporal distance between the execution of the earliest "from" entity and the latest "until" entity;
2. RepetitionRate that defines the inter-arrival time of data on a port or the triggering period of an elementary ADLFunction;
3. Input/outputSynchronization that expresses a timing requirement on the input/output synchronization among the set of ports of an ADLFunction. It should be used to express the maximum temporal skew allowed between input or output events or data of an ADLFunction.

Timing requirements specialize the meta-class TimingRestriction, which defines bounds on system timing attributes. The timing restriction can be specified as a *nominal* value, with or without a *jitter*, and can have *lower* and *upper* bounds. The jitter is the maximal positive or negative variation from the nominal value. A bound is a real value associated with an implicit time unit (ms, s...).

Example

As an illustration, we consider an Anti-lock Braking System (ABS). This example and the associated timing requirements are taken from the ATESST report on East-ADL timing model [12]. The ABS architecture consists of four sensors, four actuators and an indicator of the vehicle speed. The sensors (i_{fl}, i_{fr}, i_{rl}, i_{rr}) measure the rotation speed of the vehicle wheels. The actuators (o_{fl}, o_{fr}, o_{rl}, o_{rr}) indicate the brake pressure to be applied on the wheels. The FunctionalArchitecture is composed of FunctionalDevices for sensors and actuators and an ADLFunctionType for the functional part of the ABS. An ADLOutFlowPort provides the vehicle speed (speed).

The execution of the ABS is triggered by the occurrences of event R (Fig. 1). Parameter Ls represents the latency of sensor sampling. The values of the four sensors involved in the ABS must arrive on the input ADLFlowPorts within delay Jii (InputSynchronization). A similar OuputSynchronization delay Joo is represented on the output interface side. Lio represents the delay from the first event occurrence on the input set of the ABS until the last event occurrence on the output set. The sampling interval of the sensor is given by parameter H. All these parameters are modeled by timing requirements characterized by timing values or time intervals with jitters.

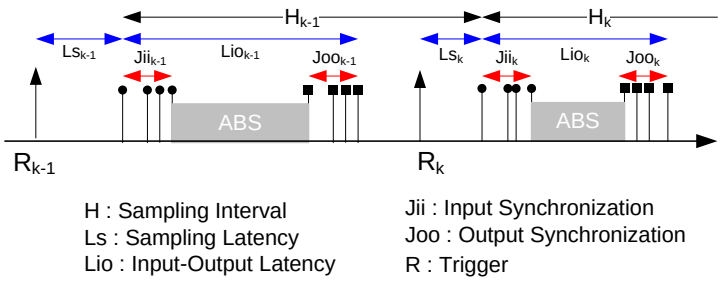

Fig. 1. Timing model of the ABS

4.2 A CCSL library for East-ADL

East-ADL introduces a vocabulary specific to the sub-domain considered (delay requirement, input/output synchronization, repetition rate). These time requirements can be modeled simply by using CCSL relations. To ease the use of such relations, user-defined relations are proposed and grouped together in a library.

Applying the UML profile for Marte

The ABS function is modeled in UML (Fig. 2) and some model elements (TimedElements) are selected to apply the CCSL clock constraints. The reaction of a timed element is dictated by the clock associated with it. For instance, the sensor i_{fl} is a timed element associated with the clock i_{fl}. When the clock i_{fl} ticks, because of the CCSL specification and the clock calculus, the sensor acquires data. Similarly when the clock o_{fl} ticks, this means that the actuator o_{fl} emits data.

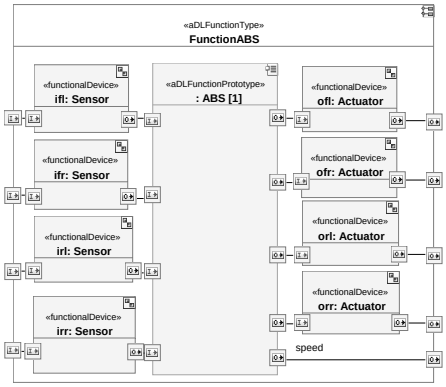

Fig. 2. Example of the ABS

In the following, we explain how the three different kinds of time requirements defined in East-ADL can be modeled with CCSL constraints.

Repetition rate

A RepetitionRate concerns successive occurrences of the same event (data arriving to or departing from a port, triggering of a function). In all cases, it consists in giving a nominal duration between two successive occurrences/instants of the same event/clock. We build a CCSL relation definition called *repetitionRate* that has three parameters: *element*, *rate* and *jitter*. *element* is the clock that must be given a repetition rate. *rate* is an integer, the actual repetition rate. *jitter* is a real number, the jitter with which the repetition rate is expressed.

$$\begin{aligned}
&\mathsf{def}\ \mathit{repetitionRate}(\mathsf{clock}\ \mathit{element},\ \mathsf{int}\ \mathit{rate},\ \mathsf{real}\ \mathit{jitter}) \triangleq \\
&\quad \mathsf{clock}\ c_1 \boxed{=} \mathit{idealClk}\ \mathsf{discretizedBy}\ 0.001 && (1)\\
&\quad |\ \mathit{element}\ \mathsf{isPeriodicOn}\ c_1\ \mathsf{period}\ \mathit{rate} && (2)\\
&\quad |\ \mathit{element}\ \mathsf{hasStability}\ \mathit{jitter}/\mathit{rate} && (3)
\end{aligned}$$

This relation definition involves three CCSL constraints. For the duration to be specified in seconds (time unit s), we use the clock *idealClk* defined in the MARTE time library (Section 3). The CCSL expression discretizedBy discretizes idealClk and defines a chronometric discrete clock c_1 so that the distance between two successive instants of c_1 is 0.001 s (Eq. 1). The unit (here s) is the default unit defined for idealClk and therefore c_1 is a 1 kHz chronometric clock. Eq. 2 uses the CCSL expression isPeriodicOn to undersample c_1 and build another clock *element*, *rate* times slower than c_1. The clock expression isPeriodicOn has not been described before but Eq. 2 is equivalent to Eq. 4.

$$\mathit{element} \boxed{=} c_1 \blacktriangledown (1.0^{rate-1})^{\omega} \tag{4}$$

Finally, Eq. 3 expresses the jitter of the repetition rate. The CCSL constraint hasStability states that the clock *element* is not strictly periodic: a maximal relative variations of *jitter*/*rate* is possible on its period.

Back to the example of the ABS. One timing requirement of the ATESST example specifies that the ABS function must be executed every 5 ms with a maximum jitter of 1 ms. If *abs.start* is the clock that triggers the execution of the function *ABS*, then $\mathit{repetitionRate}(f.\mathit{start}, 5, 1)$ enforces this requirement. A jitter of 1 ms for a nominal period of 5 ms corresponds to a stability of 20 %.

Delay requirements

A DelayRequirement constrains the delay between a set of inputs and a set of outputs. At each iteration, all inputs and outputs must occur. So, defining a delay requirement between two model elements means constraining the

temporal distance between the i^{th} occurrences of their events. In the ATESST example, a delay requirement is used, for instance to constrain the end-to-end latency for the function ABS. The definition is that at each iteration, the distance between the reception of the first input and the emission of the last output must be less than 3 ms. Consequently, we define a CCSL clock relation named *distance* that has three parameters: *begin*, *end* and *duration*. The specification is that the distance between the i^{th} occurrence of *begin* and the i^{th} occurrence of *end* must be less than *duration* ms. If we need a better precision than the ms we may define a 10 kHz chronometric clock rather than a 1kHz one (Eq. 5).

$$\begin{aligned} &\mathsf{def}\ \mathit{distance}(\mathsf{clock}\ \mathit{begin},\ \mathsf{clock}\ \mathit{begin},\ \mathsf{int}\ \mathit{duration}) \triangleq \\ &\quad \mathsf{clock}\ c_{10}\ \boxed{=}\ \mathit{idealClk}\ \mathsf{discretizedBy}\ 0.0001 \qquad (5) \\ &\quad |\ \mathit{end}\ \boxed{\prec}\ (\mathit{begin}\ \mathsf{delayedFor}\ \mathit{duration}\ \mathsf{on}\ c_{10}) \qquad (6) \end{aligned}$$

The CCSL clock expression delayedFor is a ternary operator. a delayedFor 3 on b builds a clock c that is a subclock of b. Operator *delayedFor* expresses a pure delay where the delay duration is in number of ticks of b. Note that this operator is polychronous, contrary to usual synchronous delay operators (pre in Lustre, $ in Signal). In section 6, we use the synchronous form of this operator where the third parameter is implicit (*i.e.*, a delayedFor 3 on a).

To specify the end-to-end latency, we need clocks to model the arrival of the earliest input and of the latest output. Kernel CCSL expressions inf and sup are used for that purpose.

$$\begin{array}{ll} \mathsf{clock}\ i_{inf}\ \boxed{=}\ \mathsf{inf}(i_{fl}, i_{fr}, i_{rl}, i_{rr}); & \mathsf{clock}\ i_{sup}\ \boxed{=}\ \mathsf{sup}(i_{fl}, i_{fr}, i_{rl}, i_{rr}); \\ \mathsf{clock}\ o_{inf}\ \boxed{=}\ \mathsf{inf}(o_{fl}, o_{fr}, o_{rl}, o_{rr}); & \mathsf{clock}\ o_{sup}\ \boxed{=}\ \mathsf{sup}(o_{fl}, o_{fr}, o_{rl}, o_{rr}); \end{array}$$

$inf(a, b)$ defines the greatest lower bound of a and b for the precedence relation $\boxed{\preccurlyeq}$ and $sup(a, b)$ is the lowest upper bound. With these four new clocks, stating that the end-to-end latency of the function ABS is less than 3 ms is simply written $distance(i_{inf}, o_{sup}, 30)$ in CCSL.

Similarly, input (resp. output) synchronizations are specializations of a delay requirement. An input synchronization delay requirement for the function ABS bounds the temporal distance between the earliest input and the latest input (specified by *Jii* on Fig. 1). $distance(i_{inf}, i_{sup}, 5)$ enforces an input synchronization of 0.5 ms. Likewise, an output synchronization bounds the temporal distance between the earliest output and the latest output (specified by *Joo* on Fig. 1). $distance(o_{inf}, o_{sup}, 5)$ enforces an output synchronization of 0.5 ms.

4.3 Analysis of East-ADL specification

In TimeSquare, we have implemented a specific set of menus to build East-ADL specifications. The menus give access to the different East-ADL keywords

and to the CCSL library for East-ADL. Hence, an East-ADL specification can be built interactively. Actually, the menu builds an internal model of the specification as well as a UML MARTE model. The internal model can be transformed into either a pure East-ADL model or a pure CCSL specification. The East-ADL model can be used by East-ADL-compliant tools. The CCSL specification can be analyzed by the TimeSquare clock calculus engine to detect inconsistent specifications or to execute the UML model. The execution trace can be dumped as a VCD file or can drive the animation of the UML model. Figure 3 shows an example of trace resulting from a complete specification of the ABS. This execution exhibits a violation of the specification because all the computations of the ABS itself, its sensors and actuators cannot be executed within the specified repetition rate of 5 ms. A complete analysis of this particular example is available [19].

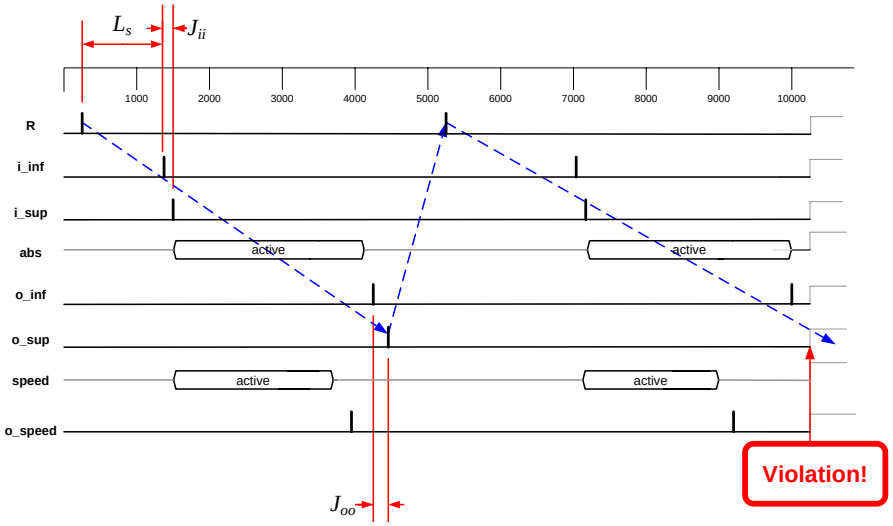

Fig. 3. The East-ADL specification of the ABS executed with TimeSquare

5 MARTE and AADL

In this second example, we consider AADL and use a combination of MARTE and CCSL to build its software components, execution platform components, express the binding relationships, and its rich model of computations and communications. Our intent is to allow a UML MARTE representation of AADL models so as UML models can benefit from the analysis tools (mainly for schedulability analysis) that accept AADL models as inputs.

5.1 AADL

Modeling elements

AADL supports the modeling of application software components (thread, subprogram, and process), execution platform components (bus, memory, processor, and device) and the *binding* of software onto execution platform. Each model element (software or execution platform) must be defined by a type and comes with at least one implementation.

The latest AADL specification acknowledge that MARTE should be used to provide a UML-based front-end to AADL models and the MARTE specification provides a full annex on the matter [7]. However, even though the annex gives full details on syntactic equivalences between MARTE stereotypes and AADL concepts, it does not say much about the semantic equivalence.

AADL application software components

Threads are executed within the context of a process, therefore the process implementations must specify the number of executed threads and their interconnections. Type and implementation declarations also provide a set of properties that characterizes model elements. For threads, AADL standard properties include the dispatch protocol (periodic, aperiodic, sporadic, background), the period (if the dispatch protocol is periodic or sporadic), the deadline, the minimum and maximum execution times, along with many others.

We have created a UML library to model AADL application software components [17]. AADL threads are modeled using the stereotype SwSchedulableResource from the MARTE Software Resource Modeling sub-profile. Its meta-attributes deadlineElements and periodElements explicitly identify the actual properties used to represent the deadline and the period. Using a meta-attribute of type Property avoids a premature choice of the type of such properties. This makes it easier for the transformation tools to be language and domain independent. In our library, MARTE type NFP_Duration is used as an equivalent for AADL type Time.

AADL flows

AADL end-to-end flows explicitly identify a data-stream from sensors to the external environment (actuators). Fig. 4 shows an example previously used [10] to discuss flow latency analysis with AADL models.

This flow starts from a sensor (Ds, an aperiodic device instance) and sinks in an actuator (Da, also aperiodic) through two process instances. The first process executes the first two threads while the last thread is executed by the second process. The two devices are part of the execution platform and communicate via a bus (db1) with two processors (cpu1 and cpu2), which host the three threads with several possible bindings. All processes are executed

by either the same processor, or any other combination. One possible binding is illustrated by the dashed arrows. The component declarations and implementations are not shown. Several configurations deriving from this example are modeled with MARTE and discussed in Section 5.2.

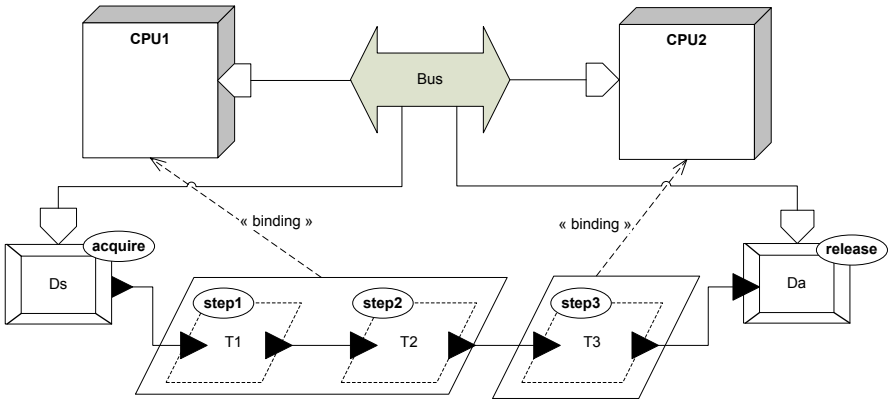

Fig. 4. The example in AADL.

AADL ports

There are three kinds of ports: *data*, *event* and *event-data*. Data ports are for data transmissions without queueing. Connections between data ports are either *immediate* or *delayed*. Event ports are for queued communications. The queue size may induce transfer delays that must be taken into account when performing latency analysis. Event data ports are for message transmission with queueing. Here again the queue size may induce transfer delays. In our example, all components have data ports represented as a filled triangle. We have omitted the ports of the processes since they are required to be of the same type as the connected port declared within the thread declaration and are therefore redundant.

UML components are linked together through ports and connectors. No queues are specifically associated with connectors. The queueing policy is better represented on a UML activity diagram that models the algorithm. A UML activity is the specification of parameterized behavior as the coordinated sequencing of actions. The sequencing is determined by *token flows*. A token contains an object, datum, or locus of control. A token is stored in an activity *node* and can move to another node through an *edge*. Nodes and edges have flow rules that define their semantics. In UML, an *object node* (a special activity node) can contain 0 or many tokens. The number of tokens in a object node can be bounded by setting its property upperBound. The order in which the tokens

present in the object node are offered to its outgoing edges can be imposed (property ordering). FIFO (First-In First-Out) is a predefined ordering value. So, object nodes can be used to represent both event and event-data AADL communication links. The token flow represents the communication itself. The standard rule is that only a single token can be chosen at a time. This is fully compatible with the AADL dequeue protocol OneItem. The UML representation of the AADL dequeue protocol AllItems is also possible. This needs the advanced activity concept of *edge weight*, which allows any number of tokens to pass along the edge, in groups at one time. The weight attribute specifies the minimum number of tokens that must traverse the edge at the same time. Setting this attribute to the unlimited weight (denoted '*') means that *all* the tokens at the source are offered to the target.

To model data ports, UML provides «datastore» object nodes. In these nodes, tokens are never consumed thus allowing multiple readings of the same token. Using a data store node with an upper bound equal to one is a good way to represent AADL data port communications.

5.2 Describing AADL models with MARTE

AADL flows with MARTE

We choose to represent the AADL flows using a UML activity diagram. Fig. 5 gives the activity diagram equivalent to the AADL example described in Fig. 4. The diagram was built with Papyrus (`http://www.papyrusuml.org`), an open-source UML graphical editor.

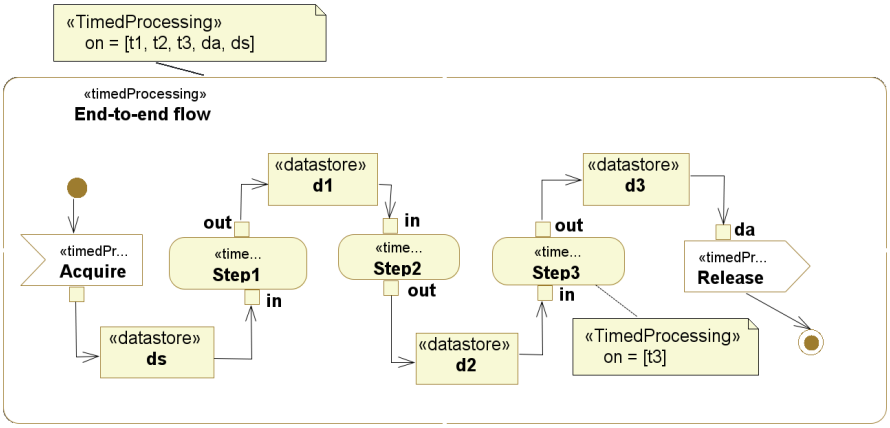

Fig. 5. End to end flow with UML and MARTE.

As discussed previously, object nodes are used to represent the queues between two tasks. This UML diagram is *untimed* and we use MARTE Time

Profile to add time information. This diagram is *a priori* polychronous since each AADL task is independent of the other tasks. The first action to describe the time behavior of this model is to build five logical clocks (*ds*, $t1$, $t2$, $t3$, *da*). This is done in two steps. Firstly, a *logical, discrete,* clock type called AADLTask is defined. Then, five instances of this clock type are built. Fig. 6 shows the final result. Secondly, the five clocks must be associated with the activity, which is done by applying the stereotype TimedProcessing. As shown in Fig. 5, this stereotype is applied to the whole activity but also to the actions. In our case, each action is associated with a different clock. In AADL, the same association is done when binding a subprogram to a task.

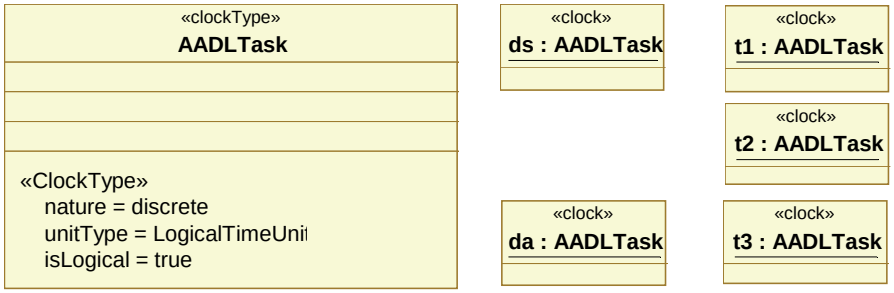

Fig. 6. One logical clock for each AADL task.

Five aperiodic tasks

The five clocks are *a priori* independent. The required time behavior is defined by applying clock constraints to these five clocks. The clock constraints to use differ depending on the dispatch protocols of the tasks. *Aperiodic* tasks start their execution when the data is available on their input port in. This is the case for devices, which are aperiodic. The *alternation* relation ($\boxed{\sim}$) can be used to model asynchronous communications. For instance, action Release starts when the data from Step3 is available in d3. $t3$ is the clock associated with Step3 and *da* is the clock associated with Release. The asynchronous communication is represented as follows: $t3 \boxed{\sim} da$. Fig. 7 represents the execution proposed by TimeSquare with only aperiodic tasks with the following constraints: $ds \boxed{\sim} t1$, $t1 \boxed{\sim} t2$, $t2 \boxed{\sim} t3$, $t3 \boxed{\sim} da$. The optional dashed arrows represent instant precedence relations induced by the applied clock constraints.

Note that this is only an abstraction of the behavior where the task durations are neglected. Additionally, we did not enforce a run to completion execution of the whole activity. Therefore, the behavior is pipelined and *ds* occurs a second time before the first occurrence of *da*. This is because the operator $\boxed{\sim}$ is not transitive. An additional constraint ($ds \boxed{\sim} da$) would be

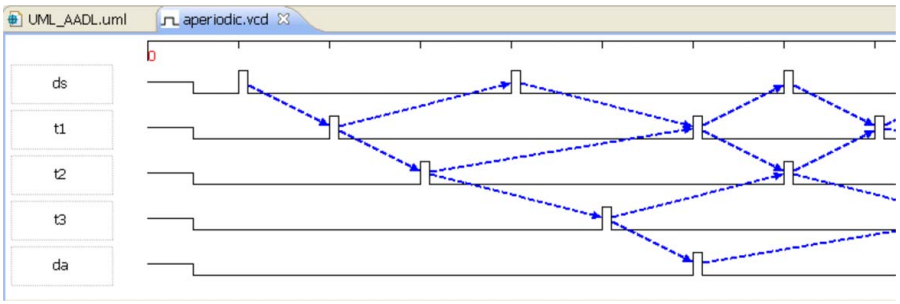

Fig. 7. Five aperiodic tasks.

required to ensure the atomic execution of the whole activity. Finally, this *run* is one possible behavior and certainly not the only one. Most of the time, and as in this case, clock constraints only impose a partial ordering on the instants of the clocks. Applying a simulation policy reduces the set of possible solutions. The one applied here is the *random policy* that relies on a pseudo-random number generator. Consequently, the result is not deterministic, but the same simulation can be replayed by restoring the generator *seed*.

Mixing periodic and aperiodic tasks

Logical clocks are infinite sets of instants but we do not assume any periodicity, *i.e.,* the distance between successive instants is not relevant. The clock constraint isPeriodicOn allows the creation of a periodic clock from another one. This is a more general notion of periodicity than the general acceptation. A clock $c1$ is said to be periodic on another clock $c2$ with period P if $c1$ ticks every P^{th} ticks of $c2$. In CCSL, this is expressed as follows: $c1$ isPeriodicOn $c2$ period P offset δ.

To build a periodic clock with the usual meaning, the base clock must refer to the physical time, *i.e.,* it must be a chronometric clock. As in Section 4, we can discretize idealClk for that purpose and build c_{100}, a 100 Hz clock (Eq. 7).

$$c_{100} \boxed{=} idealClk \text{ discretizedBy } 0.01 \tag{7}$$

Figure 8 illustrates an execution of the same application when the threads $t1$ and $t3$ are periodic. $t1$ and $t3$ are harmonic and $t3$ is twice as slow as $t1$. Coincidence instant relations imposed by the specification are shown with vertical edges with a diamond on one end. Depending on the simulation policy there may also be some opportunistic coincidences. Clock *ds* is not shown at all in this figure since it is completely independent from other clocks.

Note that, the first execution of $t3$ is synchronous with the first execution of $t1$ even before the first execution of $t2$. Hence, the task $Step3$ has no data to consume. This is compatible with the UML semantics only when using data stores. The data stores are non-depleting so if we assume an initialization step

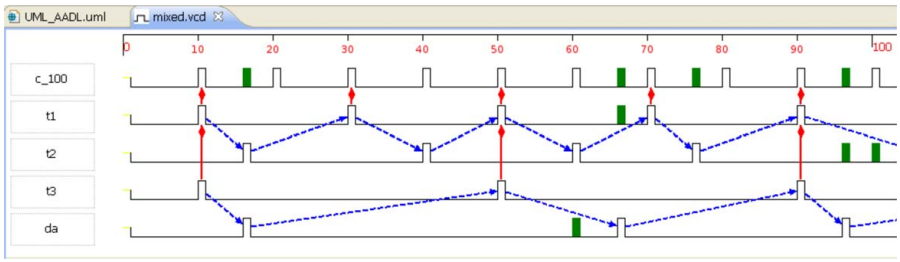

Fig. 8. Mixing periodic and aperiodic tasks.

to put one data in each data store, the data store can be read several times without any other writing. The execution is allowed, but the result may be difficult to anticipate and the same data will be read several times. When the task $t1$ is slower than $t3$, *i.e.,* when oversampling, some data may be lost.

The complete CCSL specification for this configuration is available in another work [18].

To interpret the result of the simulation, the TimeSquare VCD viewer annotates the VCD with additional information derived from the CCSL specification. We have already discussed the instant relations (dashed arrows and vertical edges). Fig. 8 also exhibits the *ghost*-tick feature. Ghosts may be hidden or shown at will and represent instants when the clock was enabled but not fired. Having a close look at clock *da*, we can see that its first occurrence is opportunistically coincident with the first occurrence of $t2$. However, the second occurrences of the two clocks are not coincident. A *ghost* is displayed (at time 60) to show that both were enabled, but $t2$ was fired alone, *da* was actually fired at the next step. In that particular example, which is not the rule, the contrary could have been true also. Additionally, that specification is *conflict-free* but it may happen that the firing of one clock disables others. These are classical problems, occurring when modeling with Petri nets, that appear with CCSL because we have defined precedence instant relations in addition to coincidence relations.

6 MARTE and SDF

This third and last example considers a purely logical case and builds a CCSL library for defining Synchronous Data Flow (SDF) [15] graphs. Subsection 6.1 recalls basics on Synchronous Data Flow (SDF) (syntax and execution rules). Section 6.2 proposes a modular CCSL specification to describe the behavioral semantics of this MoCC. Finally, an example SDF graph is built using our library. The semantics is given with CCSL and the syntax is built by a UML model with Papyrus. We apply the very same semantic model to two different UML diagrams. The first target is a UML activity, a popular notation to rep-

resent data flows. The second target is a UML state machine, whose concrete syntax is close to the usual representation of SDF graphs.

6.1 Synchronous Data Flow

Data Flow graphs are directed graphs where each node represents a function or a computation and each arc represents a data path. SDF is a special case of data flow in which the number of data samples produced and consumed by each node is specified *a priori*. This simple formalism is well suited for expressing multi-rate DSP algorithms that deal with continuous streams of data. This is a restriction of Kahn process networks [13] to allow static scheduling and ease the parallelization. SDF graphs are essentially equivalent to Computation graphs [14] which have been proven to be a special case of conflict-free Petri nets [24].

In SDF graphs, nodes (called *actors*) represent operations. Arcs carry *tokens*, which represent data values (of any data type) stored in a first-in first-out queue. Actors have inputs and outputs. Each input (resp. output) has a *weight* that represents the number of tokens consumed (resp. produced) when the actor executes. SDF graphs obey the following rules:

- An actor is *enabled* for execution when all inputs are enabled. An input is enabled when enough tokens are available on the incoming arc. Enough tokens means equal to or greater than the input weight. Actors with no inputs (Source actors) are always enabled. Enabling and execution never depend on the token values, *i.e.,* the control is data-independent.
- When an actor executes, it always produces and consumes the same fixed amount of tokens, in an atomic way. It produces on each output exactly the number of tokens specified by the output weight; these tokens are written into the queue of the outgoing arc. It consumes on each input exactly the number of tokens specified by the input weight, these tokens are read (and removed) from the queue of the incoming arc.
- *Delay* is a property of an arc. A delay of n samples means that n tokens are initially in the queue of the arc.

6.2 A CCSL library for SDF

With SDF graphs a local observation is enough to know the dependency on a given element. So it is possible to construct locally a set of CCSL constraints for each model element. This section describes the library of CCSL relations built for this purpose.

As illustrated in previous examples, the first stage is to identify which CCSL clocks must be defined to create a time system conforming to the SDF semantics. For each *actor* A, one CCSL clock A is created. The instants of this clock represent the atomic execution instants of the operation related to the actor. For each *arc* T, two CCSL clocks *write* and *read* are created. Clock

write ticks whenever a token is written into the arc queue. Clock *read* ticks whenever a token is read (and removed) from the queue. Note that, the actual number of available tokens is not directly represented and must be computed, if required, from the difference in the number of read and write operations. No specific clocks are created for inputs and outputs.

The second stage is to apply the right CCSL clock constraints so that the result of the clock calculus can be interpreted to apprehend the behavioral semantics of the SDF graph.

Actors

do not require any specific constraint.

Tokens

are written into and read from the arc queues. A *token* cannot be read before having been written, hence, for a given arc, the i^{th} tick of *write* must strictly precede the i^{th} tick of *read*. The kernel CCSL relation strict precedence models such a constraint: $write \boxed{\prec} read$. When $delay > 0$, $delay$ tokens are initially available in the queue, which means that the i^{th} read operation gets the data written at the $(i - delay)^{\text{th}}$ write operation, for $i > delay$. The $delay$ previous read operations actually get tokens initially available and that do not match an actual write operation. CCSL operator delayedFor can represent such a delay (Eq. 8). To represent SDF arcs, we propose to create in a library, a new relation definition, called *token*. Such a relation has three parameters: two clocks (*write* and *read*) and an integer (*delay*). The *token* relation applies the adequate constraint (Eq. 8). Note that, when $delay = 0$, Eq. 8 reduces to $write \boxed{\prec} read$.

$$\begin{aligned} &\mathsf{def}\ token(\mathsf{clock}\ write,\ \mathsf{clock}\ read,\ \mathsf{int}\ delay) \triangleq \\ &\quad write \boxed{\prec} (read\ \mathsf{delayedFor}\ delay) \end{aligned} \tag{8}$$

Inputs

require a packet-based precedence (keyword by in Eq. 9). A relation definition, called *input*, has three parameters. The clock *actor* represents the actor with which the input is associated. The clock *read* represents the reading of tokens on the incoming arc. The strictly positive integer *weight* represents the input weight.

$$\begin{aligned} &\mathsf{def}\ input(\mathsf{clock}\ actor,\ \mathsf{clock}\ read,\ \mathsf{int}\ weight) \triangleq \\ &\quad (read\ \mathsf{by}\ weight) \boxed{\prec} actor \end{aligned} \tag{9}$$

Here again, the packet-based precedence can be built with the filtering operator (Eq. 10). When $weight = 1$, it reduces to $read \boxed{\prec} actor$.

$$\left(read \blacktriangledown \left(0^{weight-1}.1\right)^{\omega}\right) \boxed{\prec} actor \tag{10}$$

Outputs

are represented by the relation definition *output*, which has three parameters. The clock *actor* represents the actor with which the output is associated. The clock *write* represents the writing of tokens to the outgoing arc. The strictly positive integer *weight* represents the output weight. CCSL operator filteredBy is used (Eq. 11). When $weight = 1$, Eq. 11 simplifies into $actor \boxed{=} write$.

$$\begin{aligned} &\mathsf{def}\ output(\mathsf{clock}\ actor,\ \mathsf{clock}\ write,\ \mathsf{int}\ weight) \triangleq \\ &\quad actor \boxed{=} \left(write \blacktriangledown \left(1.0^{weight-1}\right)^{\omega}\right) \end{aligned} \tag{11}$$

Arcs

can globally be seen as a conjunction of one output, one token and one input. The library relation definition *arc* (Eq. 12) specifies a complete set of constraints for an arc with a delay *delay*, from an actor *source*, with an output weight *out* to another actor *target*, with an input weight *in*. In that case, CCSL clocks *write* and *read* are local clocks.

$$\begin{aligned} &\mathsf{def}\ arc(\mathsf{int}\ delay,\ \mathsf{clock}\ source,\ \mathsf{int}\ out,\ \mathsf{clock}\ target,\ \mathsf{int}\ in) \triangleq \\ &\quad \mathsf{clock}\ read, write \\ &\quad output(source, write, out) \\ &\quad |\ token(write, read, delay) \\ &\quad |\ input(target, read, in) \end{aligned} \tag{12}$$

6.3 Applying the SDF semantics

The previous subsection defines, for each SDF model element, the CCSL clocks that must be created and the clock constraints to apply. This section illustrates the use of our CCSL library for SDF. Our purpose is to explicitly add to an existing model the SDF semantics. In this example, we use UML activities (Fig. 9.a) and state machines (Fig. 9.c) to build with the Papyrus editor a simple SDF graph (Fig. 9.b). Our intent is NOT to extend the semantics of UML activities and state machines but rather to use Papyrus as a mere graphical editor to build graphs, *i.e.,* nodes connected by arcs. Our proposal is to attach CCSL clocks to some UML elements represented by the modeling

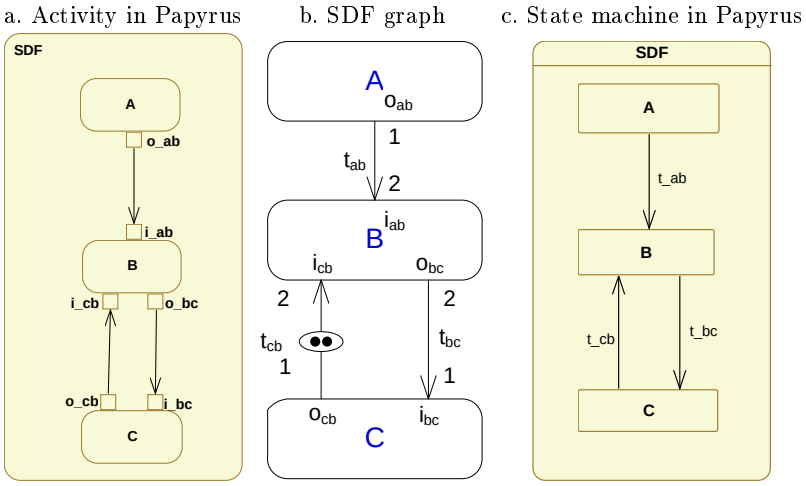

Fig. 9. An example of SDF graph

editor. Papyrus gives the concrete graphical syntax and the clock constraints give explicitly the expected execution rules.

By instantiating elements of our CCSL library for SDF, these two models can be given the execution semantics as classically understood for SDF graphs. The idea is to add the semantics within the model explicitly without being implicitly bound to some external description. In our case, the semantics is given by the CCSL specification and by explicit associations between CCSL clocks and model elements.

When using the syntax of activities, CCSL clocks that represent actors are associated with actions and CCSL clocks that represent readings from (resp. writings to) the arc queues are associated with input (resp. output) pins. All other CCSL clocks are left unbound and are just used as local clocks with no direct interpretation on the model. When using the syntax of state machines, only CCSL clocks that represent actors are bound to the states and other clocks are left unbound.

Eq. 13 uses our library to give to the models on parts (a) and (c) the same semantics as understood when considering the SDF graph in the middle part (b). Clocks are named after the model elements with which they are associated, *i.e.,* the clock for actor A is named A. However, this rule is for clarity only and the actual association is done explicitly in the CCSL model.

$$\mathcal{S} \triangleq arc(0, A, 1, B, 2) \mid arc(0, B, 2, C, 1) \mid arc(2, C, 1, B, 2) \tag{13}$$

Fig. 10 shows one possible execution of this specification produced by TimeSquare. Intermediate clocks are hidden and only actors are displayed. The relative execution of the actors is what matters when considering SDF graphs.

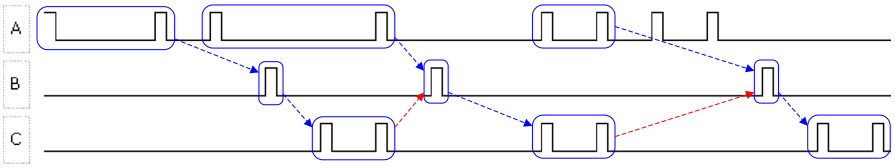

Fig. 10. A simulation output with TimeSquare

7 Conclusion

The UML profile for MARTE extends the UML to model real-time and embedded systems, at the appropriate level of description. However, the real-time and embedded domain is vast and lots of efforts have already been done to provide dedicated proprietary UML extensions [5, 11, 25]. MARTE clearly does not cover the whole domain and should be refined to tackle specific aspects, and provide a larger support for analysis and synthesis tools. Our contribution is to illustrate the use of logical time for system specification. This is done by using the MARTE time subprofile conjointly with CCSL. Logical time is thus an integral part of the functional design that can be exploited by compilation/synthesis tools, not restricting time to annotations for performance evaluation or simulation.

At the same time, there is a large demand for standards because the community, tool vendors and manufacturers, wants to rely on industry-proven, perennial languages. Many specifications in various subdomains are issued by various organizations to answer this demand (AADL, AUTOSAR, East-ADL, IP-Xact ...), even though these subdomains have covered for a long time separate markets and were addressed by different communities. With the emergence of large systems, systems of systems, we need to combine several aspects of these subdomains into a common modeling environment. The automotive and avionic industries, which were using dedicated processors are now integrating generic processors and need interoperability between their own models and models used by electronic circuits manufacturers. Therefore, we need formalisms able to compose these models both syntactically and semantically. Having a common semantic framework is required to ensure interoperability of tools. By selecting several examples from some of these subdomains, we have shown that MARTE time profile and CCSL can be used to tackle different aspects of some of these emerging specifications. We advocate that the composed model must provide structural composition rules but also a description of the intent semantics, so that some analysis can be done at the model level. Then, this model can serve as a golden specification for equivalence comparison with other formal models suitable to apply specific analysis or synthesis techniques.

References

1. C. André. Syntax and Semantics of the Clock Constraint Specification Language (CCSL). Research Report 6925, INRIA, May 2009.
2. A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.
3. A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. N-synchronous Kahn networks: a relaxed model of synchrony for real-time systems. In J. G. Morrisett and S. L. P. Jones, editors, *POPL*, pages 180–193. ACM, January 2006.
4. P. Cuenot, D. Chen, S. Gérard, H. Lönn, M.-O. Reiser, D. Servat, C.-J. Sjostedt, R. T. Kolagari, M. Torngren, and M. Weber. Managing complexity of automotive electronics using the East-ADL. In *Proc. of the 12th IEEE Int. Conf. on Engineering Complex Computer Systems (ICECCS'07)*, pages 353–358. IEEE Computer Society, 2007.
5. B. P. Douglass. Real-time UML. In W. Damm and E.-R. Olderog, editors, *FTRTFT*, volume 2469 of *Lecture Notes in Computer Science*, pages 53–70. Springer, 2002.
6. J. Eker, J. Janneck, E. Lee, J. Liu, X. L., J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.
7. M. Faugère, T. Bourbeau, R. de Simone, and S. Gérard. Marte: Also an UML profile for modeling AADL applications. In *ICECCS - UML&AADL*, pages 359–364. IEEE Computer Society, 2007.
8. P. Feautrier. Compiling for massively parallel architectures: a perspective. *Microprogramming and Microprocessors*, (41):425–439, 1995.
9. P. Feiler, D. Gluch, and J. Hudak. The architecture analysis & design language (AADL): An introduction, 2006.
10. P. H. Feiler and J. Hansson. Flow latency analysis with the architecture analysis and design language. Technical Report CMU/SEI-2007-TN-010, CMU, June 2007.
11. S. Graf. OMEGA: correct development of real time and embedded systems. *Software and System Modeling*, 7(2):127–130, 2008.
12. R. Johansson, H. Lönn, and P. Frey. ATESST timing model. Technical report, ITEA, 2008. Deliverable D2.1.3.
13. G. Kahn. The semantics of a simple language for parallel programming. *Information Processing*, pages 471–475, 1974.
14. R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, 1966.
15. E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Computers*, 36(1):24–35, 1987.
16. E. A. Lee and A. L. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.
17. S.-Y. Lee, F. Mallet, and R. de Simone. Dealing with AADL end-to-end flow latency with UML Marte. In *ICECCS - UML&AADL*, pages 228–233. IEEE CS, April 2008.

18. F. Mallet and R. de Simone. *MARTE vs. AADL for Discrete-Event and Discrete-Time Domains*, volume 36 of *LNEE*, chapter 2, pages 27–41. Springer, April 2009.
19. F. Mallet, M.-A. Peraldi-Frati, and C. André. Marte CCSL to execute East-ADL timing requirements. In *ISORC*, pages 249–253. IEEE Computer Society, March 2009.
20. OMG. *UML Profile for Schedulability, Performance, and Time specification, v1.1.* Object Management Group, January 2005. formal/05-01-02.
21. OMG. *Systems Modeling Language (SysML) Specification, v1.1.* Object Management Group, November 2008. formal/08-11-02.
22. OMG. *UML Profile for MARTE, v1.0.* Object Management Group, November 2009. formal/2009-11-02.
23. OMG. *Unified Modeling Language, Superstructure, v2.2.* Object Management Group, February 2009. formal/2009-02-02.
24. C. Petri. Concurrency theory. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and their properties*, volume 254 of *Lecture Notes in Computer Science*, pages 4–24. Springer-Verlag, 1987.
25. B. Selic. The emerging real-time UML standard. *Comput. Syst. Sci. Eng.*, 17(2):67–76, 2002.
26. S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors Scheduling and Synchronization.* CRC Press, Taylor & Francis Group, Boca Raton (FL), second edition, 2009.
27. The ATESST Consortium. East-ADL2 specification. Technical report, ITEA, March 2008. http://www.atesst.org, 2008-03-20.
28. The East-EEA Project. Definition of language for automotive embedded electronic architecture approach. Technical report, ITEA, 2004. Deliverable D.3.6.
29. T. Weilkiens. *Systems Engineering with SysML/UML: Modeling, Analysis, Design.* The MK/OMG Press, Burlington, MA, USA., 2008.