Executing AADL models with UML/Marte

Frédéric Mallet, Charles André, Julien Deantoni

▶ **To cite this version:**

Frédéric Mallet, Charles André, Julien Deantoni. Executing AADL models with UML/Marte. Int. Conf. Engineering of Complex Computer Systems - ICECCS'09, Jun 2009, Potsdam, Germany. pp. 371-376, 10.1109/ICECCS.2009.10 . inria-00416592

HAL Id: inria-00416592
https://inria.hal.science/inria-00416592

Submitted on 14 Sep 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ICECCS - UML&AADL'09. Potsdam, Germany. June 2009, pp. 371–376

Executing AADL models with UML/MARTE

Frédéric Mallet, Charles André, Julien DeAntoni
Aoste Team-Project
Université de Nice Sophia Antipolis
INRIA Sophia Antipolis Méditerranée
Email: {fmallet,candre,jdeanton}@sophia.inria.fr

Abstract

AADL and MARTE are two modeling formalisms supporting the analysis of real-time embedded systems. Since both cover similar aspects, a clear assessment of their respective strength and weakness is required. Building on previous works, we focus here[1] *on the time aspects of the two specifications. Relying on the MARTE Time Model and the operational semantics of its companion language CCSL we attempt to equip UML activities with the execution semantics of an AADL specification. This is part of a much broader effort to build a generic simulator for UML models with the semantics explicitly defined within the model.*

1. Introduction

AADL and MARTE are two frameworks that support modeling and analysis of embedded systems. Both provide model elements to represent the application (logical, platform-independent solution), the architecture (execution platform) and to define the mapping of application functions onto architecture resources and services. AADL (Architecture Analysis & Design Language) [1] was developed as a standard of the Society of Automotive Engineers (SAE), whereas MARTE (Modeling and Analysis of Real Time and Embedded systems) [2] is a UML profile recently adopted by OMG (Object Management Group).

Despite their many similar features, they differ in several points. AADL focuses on analysis models and some transformations are provided to feed AADL models into schedulability analysis tools like Cheddar [3]. MARTE is more generic on the modeling aspects and attempts to have a much broader scope. Being a UML profile, it benefits from the large set of rapidly improving UML graphical editors and has strong connections with SysML, offering an opportunity to have a complete and integrated design flow, from system-level specification and requirement analysis to implementation, code generation, schedulability and performance analysis. Using MARTE as a foundation profile to model AADL-specific concepts would benefit to both communities. It would ease interoperability of AADL with UML models and would allow some selected MARTE models to be analyzed by AADL tools.

There have been continuous efforts to establish a strong relationship between the two specifications [4], [5], [6], [7]. We have previously discussed systematic model transformations from AADL to MARTE focusing on the analysis of end-to-end flow latencies. We are discussing here the possibility to execute AADL specifications by relying on the MARTE time model and on the operational semantics of its companion language CCSL (Clock Constraint Specification Language) [8]. This effort should lead to the construction of a generic UML simulation engine able to execute AADL specifications with the semantics *explicitly built* as part of the model.

Our approach clearly departs from the work on ADeS (available within TopCased, http://www.topcased.org) to build a simulator specifically for AADL. It also differs from the methodology used in other transformation approaches (like AADL2sync [9], [10]). Transformation from AADL models into other formalisms equipped with a formal operational semantics is useful to support both simulation and formal verification. The transformation [9] changes AADL architectures into Lustre, a purely synchronous language. It emulates the AADL asynchronism using *quasi-synchronous* communications. The second [10] uses Signal, as a target language. Signal being a polychronous language, asynchronous communications are natively supported. However, in both cases, this is totally separate from UML and more importantly the actual semantics remains within the transformation tools whereas we intend to make it explicit within the UML model.

Section 2 browses through some AADL concepts required for understanding the remaining of the paper. A more detailed description have already been presented [7]. Section 3 introduces TIMESQUARE, the analysis environment built to implement and support the MARTE time model. Section 4 implements the AADL example with UML and MARTE and illustrates the simulation facilities offered by TIMESQUARE to execute the models under different configurations.

1. The original publication is available at http://dx.doi.org/10.1109/ICECCS.2009.10/

©IEEE 2009

2. AADL

2.1. 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.

Initially, there were plans to create a specific UML profile for AADL. However, due to the existence of the OMG UML profile for MARTE, a new OMG UML profile for AADL is unlikely to be adopted [4]. Instead, the MARTE specification provides guidelines for UML representation of AADL models.

2.2. AADL application software components

Threads are executed within the context of a process, therefore the process implementations must specify the number of threads they execute 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 [5]. 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.

2.3. AADL flows

AADL end-to-end flows explicitly identify a data-stream from sensors to the external environment (actuators). Figure 1 shows an example previously used [11] 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 processes 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 4.

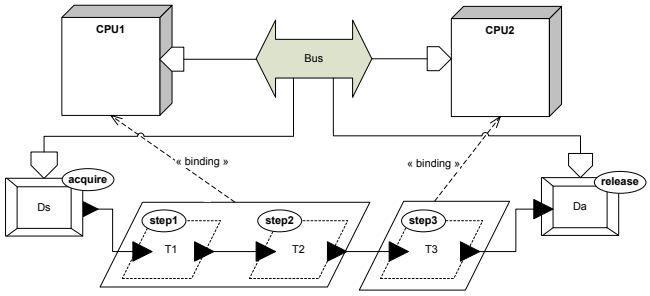

Figure 1. The example in AADL.

2.4. 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 than 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.

3. The MARTE Time Model and CCSL

This section only briefly introduces MARTE Time Model. More details can be found in another paper [8].

3.1. Clock and Time Structure

A *Clock* is an ordered set of instants ($\mathcal{I}$), where $\prec$ is a quasi-order relation on $\mathcal{I}$, named *strict precedence*.

A *discrete-time clock* c is a clock with a discrete set of instants $\mathcal{I}$. Since $\mathcal{I}$ is discrete, it can be indexed by natural numbers in a fashion that respects the ordering on $\mathcal{I}$. $c[k]$ denotes the k^{th} instant. Moreover, in the discrete case, each instant, but the first one, has a unique direct predecessor.

A set of clocks constrained by *clock constraints* defines a *Time Structure*. More formally, a time structure is a pair $\langle C, \preccurlyeq \rangle$ where C is a set of clocks, $\preccurlyeq$ is a binary relation on $\bigcup_{c \in C} \mathcal{I}_c$, named *precedence*. $\preccurlyeq$ is reflexive and transitive. From $\preccurlyeq$ we derive four new instant relations: *Coincidence* ($\equiv \triangleq \preccurlyeq \cap \succcurlyeq$), *Strict precedence* ($\prec \triangleq \preccurlyeq \setminus \equiv$), *Independence* ($\| \triangleq \overline{\preccurlyeq \cup \succcurlyeq}$), and *Exclusion* ($\# \triangleq \prec \cup \succ$).

Instant relations are defined on pairs of instants. This is obviously not suitable for a time structure specification. Instead we have defined constraints on clocks: a clock constraint imposes many—usually infinitely many—instant constraints.

3.2. Clock constraints

Clock constraints can be divided into four categories: *synchronous*, *asynchronous*, *mixed*, and *NFP chronometric* (NFP stands for Non Functional Properties).

Synchronous clock constraints rely on *coincidence*. *Subclocking* is such a constraint: each instant of the *subclock* must coincide with one instant of the *superclock*. Of course, the mapping must be order-preserving.

Asynchronous clock constraints are based on *precedence*. A (discrete) clock a *strictly precedes* clock b if for all natural number k, the k^{th} instant of a precedes the k^{th} instant of b ($\forall k \in \mathbb{N}^\star, a[k] \prec b[k]$).

Mixed clock constraints combine both coincidence and precedence. For instance the *sampling* constraint: $c = a$ sampledOn b imposes c to tick synchronously with b whenever a tick of a precedes a tick of b.

NFP chronometric constraints apply to chronometric clocks. They specify temporal properties such as *stability*, *offset*, *jitter*, etc. These constraints are used to characterize imperfect chronometric clocks.

Semantics of clock constraints. A *Time Structure* is considered as a dynamic system and its behavior is defined by an infinite sequence of *steps*. A step consists of simultaneous clock ticks. When a (discrete) clock ticks, its current instant changes for the next one (its current index is incremented by 1). We call *configuration* of a time structure $\langle C, \preccurlyeq \rangle$ a mapping $c : C \to \mathbb{N}$. For each discrete clock clk, $c(clk)$ is the current index of clock clk. This index denotes the *current instant* of clk.

For a set of clocks subject to a conjunction of clock constraints, the challenge is, "given a configuration, determine a step that meets all the constraints". There may be 0 (inconsistent constraints), 1 (deterministic) or several satisfying steps (non deterministic).

To address this challenge, we have endowed CCSL with a structural operational semantics. It is sufficient to define SOS rules for a *kernel* CCSL (less than 20 rules) [12]. For illustration purpose, consider the "strictly precedes" relation $c_1 \boxed{\prec} c_2$.

$$\frac{\begin{array}{l} c_1, c \vdash \mathrm{b}_1 \\ c_2, c \vdash \mathrm{b}_2 \\ \mathrm{b} \triangleq (c(c_1) = c(c_2)) \end{array}}{c_1 \boxed{\prec} c_2, c \vdash \mathrm{b}_1 \wedge \mathrm{b}_2 \wedge (\mathrm{b} \Rightarrow \neg \mathrm{c}_2)}$$

This rule reads that, for the given configuration c, constraint "c_1 strictly precedes c_2" implies the Boolean expression on the right-hand side. In this rule, $\mathrm{c_k}$ is a Boolean variable associated with clock c_k. $\mathrm{c_k} = \mathsf{true}$ means that c_k can tick. The Boolean expression refers to Boolean expressions (b_1, b_2) attached to the concerned clocks (c_1, c_2), and imposes additional logical constraints, specific to the precedence relation: $(\mathrm{b} \Rightarrow \neg \mathrm{c}_2)$ or equivalently $(\neg \mathrm{b} \vee \mathrm{c}_2)$.

Clock constraints not defined in the kernel CCSL can be specified in terms of primitive clock constraints. Some usual constraints are predefined. The *alternation* (denoted by $\boxed{\sim}$) used in Section 4.2 is a case in point. $a \boxed{\sim} b$ imposes that $\forall k \in \mathbb{N}^\star, a[k] \prec b[k] \prec a[k{+}1]$. This is defined in CCSL by $\big(a \boxed{\prec} b\big) \,\|\, \big(b \boxed{\prec} a[2..*]\big)$, where $a[2..*]$ is clock a deprived of its first instant.

From a CCSL specification we derive a set of Boolean expressions. Let B be the conjunction of all these expressions. Starting with B, we determine the set of all

possible (logical) solutions. From this set we deduce the set E of *Enabled Clocks*. A subset F (*Fired Clocks*) of E characterizes the new *step*. Not all subsets of E are correct solutions because a step must contain all or none of the clocks that have coincident instants. To derive F from E, the user chooses among different policies: minimal solution, maximal solution, random selection, and user's defined policies. The default policy is the random selection which chooses one out of all the correct solutions.

Applying *rewriting rules* of the form $c_1 \xrightarrow{c_1 \in F} c_1'$ for all fired clocks yields the new set of clock constraints.

3.3. TimeSquare Environment

TIMESQUARE is the software environment we have developed to support the MARTE Time model and the analysis of clock constraint specifications.

TIMESQUARE has four main features: 1) interactive clock-related specifications, 2) clock constraint checking, 3) generation of a solution, 4) displaying and exploring waveforms.

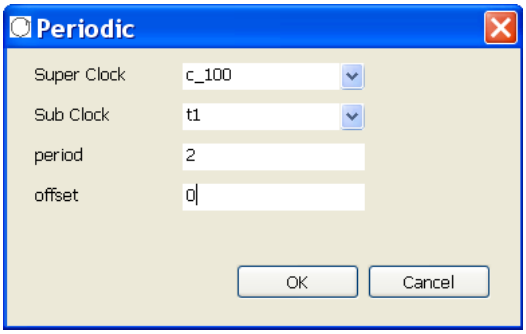

Figure 2. Dialog box for clock constraints.

TIMESQUARE has been designed to be used with UML tools applying the MARTE profile. In this profile, clocks and clock constraints can be associated with many and various model elements. A wizard is included in TIMESQUARE. It facilitates clock definitions, clock constraint specifications, model element browsing, and parameter setting. Figure 2 shows an example of dialog box for specifying clock constraints.

The second feature checks constraint sanity and is called when the above mentioned wizard is not used.

The third feature relies on a *constraint solver* that yields a satisfying execution trace or issues an error message in case of inconsistency. The traces are given as waveforms written in VCD format. VCD (Value Change Dump) [13] is an IEEE standard textual format for dumpfiles used by EDA (Electronic Design Automation) logic simulation tools. The solver intensively uses Binary Decision Diagrams (BDD).

Waveforms can be displayed with any VCD viewer. TIMESQUARE has its own viewer enriched with interactive constraint highlighting and access facilities.

3.4. Implementation

TIMESQUARE is a collection of plug-ins developed with Ganymede Eclipse Modeling Tools (Eclipse packaging including EMF, GMF, MDT XSD/OCL/UML2, M2M, M2T, and EMFT). ANTLR for constraint parsing, and JavaBDD for the solver are also used. TIMESQUARE is integrated in the OpenEmbeDD platform (http://openembedd.org) and is available for download at http://www.inria.fr/sophia/aoste/dev/time_square.

4. Execution of the AADL example

4.1. AADL flow with MARTE

We choose to represent the AADL flow using a UML activity diagram. Figure 3 gives the activity diagram equivalent to the AADL example described in Figure 1. The diagram was built with Papyrus (http://www.papyrusuml.org), an open-source UML graphical editor.

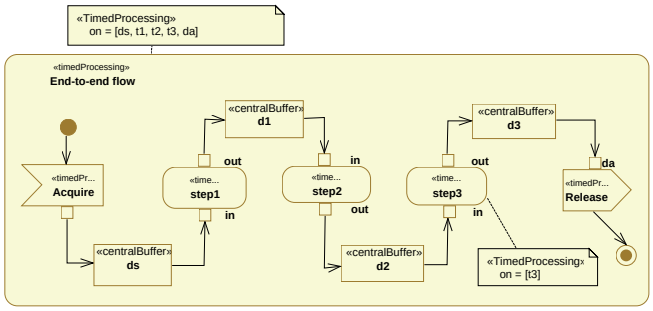

Figure 3. End to end flow with UML and MARTE.

As discussed in subsection 2.4, object nodes are used to represent the queues between two tasks. We can either use CentralBufferNode for event and event-data ports or DataStore for data ports. The upper bound fixes the length of the queues when required. This UML diagram is *untimed*, to make it behave the exact same way as an AADL model, we propose to use the MARTE Time Profile. 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. Figure 4 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 Figure 3, 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.

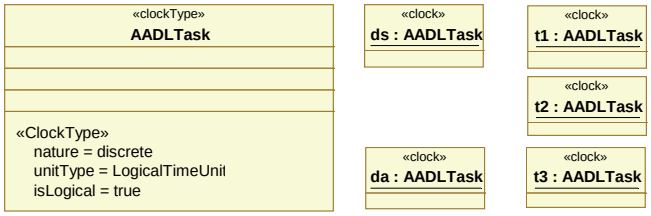

Figure 4. One logical clock for each AADL task.

4.2. 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* constraint 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: $t3 \boxed{\sim} da$. Figure 5 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.

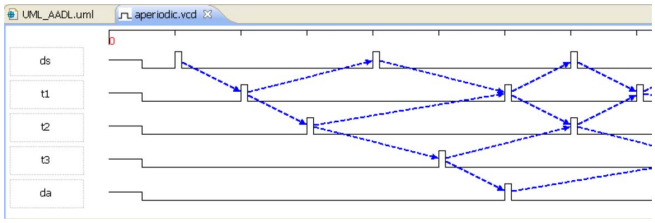

Figure 5. Five aperiodic tasks.

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 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 its *seed*.

4.3. 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 known. 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. The relation discretizedBy is used to discretize idealClk, a dense chronometric (related to physical time) assumed to be a perfect clock (with no jitter or any other flaw). Eq. 1 creates, as an example, a 100 Hz clock.

$$c_{100} = idealClk \text{ discretizedBy } 0.01 \quad (1)$$

Eq. 1 states that the distance (duration) between two successive instants of clock c_{100} is 0.01 s. The unit second (s) is implied by the use of idealClk.

Figure 6 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 side. 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.

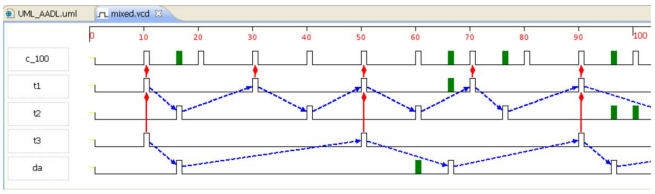

Figure 6. Mixing periodic and aperiodic tasks.

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 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 given in Eqs. 2.

$$\begin{array}{l} c_{100} = idealClk \text{ discretizedBy } 0.01; \\ t1 \text{ isPeriodicOn } c_{100} \text{ period } 2 \text{ offset } 0; \\ t1 \text{ alternatesWith } t2; \\ t3 \text{ isPeriodicOn } t1 \text{ period } 2 \text{ offset } 0; \\ t3 \text{ alternatesWith } da; \end{array} \quad (2)$$

As pointed out by Feiertag and *al.* [14], over and under-sampling issues are frequent in multi-rate, register-based systems. These authors have introduced different semantics for the end-to-end delay. The *sampling* constraint present in CCSL can simulate these semantics. However, we have not dealt here with all those semantic variations.

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). Figure 6 also exhibits *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 ds, 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 to show that both were enabled, but $t2$ was fired first and alone, ds was actually fired at the following 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 and that appear with CCSL because we have defined precedence instant relations in addition to coincidence relations.

5. Conclusion

AADL and MARTE have many similar features but also differ on many aspects. This paper continues our effort to compare them and specifically focuses on the possibility to execute AADL specifications by relying on MARTE time model and on the operational semantics of CCSL. We discuss the representation of AADL periodic and aperiodic tasks. We also discuss the different CCSL constraints that must be applied to give a standard UML activity diagrams the same execution semantics than an AADL model. The simulation executions are displayed as waveforms but we have also implemented an animator for Papyrus. This animator uses the operational semantics of CCSL constraints to decide which clocks must tick. The clocks are associated with UML model elements. When a given clock is fired, the related DI2 *graph element* associated with the related model element is modified. DI2 is the diagram interchange format defined by the OMG and implemented by Papyrus. Different graph elements are modified differently. For instance, actions and states have their background color modified, whereas transitions and activity flows are highlighted.

References

[1] SAE, *Architecture Analysis and Design Language (AADL)*, June 2006, aS5506/1, http://www.sae.org.

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

[3] F. Singhoff and A. Plantec, "AADL modeling and analysis of hierarchical schedulers," in *SIGAda*, A. Srivastava and L. C. B. III, Eds. ACM, 2007, pp. 41–50.

[4] M. Faugère, T. Bourbeau, R. de Simone, and S. Gérard, "Marte: Also an UML profile for modeling AADL applications," in *ICECCS - UML&AADL*. IEEE Computer Society, 2007, pp. 359–364.

[5] S.-Y. Lee, F. Mallet, and R. de Simone, "Dealing with AADL end-to-end flow latency with UML Marte," in *ICECCS - UML&AADL*. IEEE CS, April 2008, pp. 228–233.

[6] C. André, F. Mallet, and R. de Simone, *Modeling of AADL data-communications with UML Marte*, ser. LNEE. Springer, May 2008, vol. 10, ch. 11, pp. 150–170.

[7] F. Mallet, R. de Simone, and L. Rioux, "Event-triggered vs. time-triggered communications with UML Marte," in *FDL*. IEEE, 2008, pp. 154–159.

[8] F. Mallet, "CCSL: specifying clock constraints with UML/Marte," *ISSE*, vol. 4, no. 3, pp. 309–314, 2008.

[9] E. Jahier, N. Halbwachs, P. Raymond, X. Nicollin, and D. Lesens, "Virtual execution of aadl models via a translation into synchronous programs," in *EMSOFT*, C. M. Kirsch and R. Wilhelm, Eds. ACM, 2007, pp. 134–143.

[10] M. Yue, J.-P. Talpin, and T. Gautier, "Virtual prototyping aadl architectures in a polychronous model of computation," in *MEMOCODE*. IEEE Computer Society, 2008, pp. 139–148.

[11] P. H. Feiler and J. Hansson, "Flow latency analysis with the architecture analysis and design language," CMU, Tech. Rep. CMU/SEI-2007-TN-010, June 2007.

[12] C. André and F. Mallet, "Combining CCSL and Esterel to specify and verify time requirements," INRIA, Research Report RR-6839, 2009. [Online]. Available: http://hal.inria.fr/inria-00360528/en/

[13] IEEE Standards Association, *IEEE Standard for Verilog Hardware Description Language*, Design Automation Standards Committee, 2005, IEEE Std 1364TM-2005.

[14] N. Feiertag, K. Richter, J. Nordlander, and J. Jonsson, "A compositional framework for end-to-end path delay calculation of automotive sytems under different path semantics," Work. on Compositional Theory and Technology for Real-Time Embedded Systems CRTS, Barcelona (E), 2008.