Dealing with AADL End-to-end Flow Latency with UML MARTE

Su-Young Lee, Frédéric Mallet, Robert de Simone

N° 6402

Janvier 2008

Thème COM

ISSN 0249-6399

INRIA Research report

Su Young Lee[1], Frédéric MALLET[2], Robert de Simone[3]

Thème Com – Systèmes communicants
Equipe-Projet Aoste

Rapport de recherche n° **6402** – Janvier 2008 - 21 pages

Abstract: AADL and MARTE are both modeling formalisms supporting the analysis of real-time embedded systems. We investigate how MARTE, with its Time Model facilities, can be made to represent faithfully AADL periodic/aperiodic tasks communicating through event or data ports, in an approach to end-to-end flow latency analysis.

Keywords: AADL, UML MARTE, latency analysis, MoCC.

[1] INRIA Sophia – Su-Young.Lee@sophia.inria.fr
[2] Université de Nice Sophia Antipolis & INRIA Sophia – fmallet@unice.fr
[3] INRIA Sophia – Robert.de_Simone@sophia.inria.fr

INRIA Research Report

Résumé: AADL et MARTE sont deux formalismes qui supportent l'analyse de systèmes temps-réel embarqués. Nous avons étudié la capacité de MARTE, et notamment de son modèle de temps, à représenter fidèlement les mécanismes de communications fournis par AADL. Nous nous sommes intéressés plus particulièrement à son mécanisme de tâches périodiques et apériodiques communicantes au travers de port d'événements, de données ou mixte. Cette étude est appliquée à un exemple d'analyse de latence de bout en bout qui est un cas d'étude classique utilisé par AADL.

Mots clés: AADL, MARTE, analyse de latence de bout en bout, MoCC.

1 Introduction

Modern embedded systems must support the deployment of heterogeneous applications onto heterogeneous architectures. At design time the targeted execution platform may still be speculative, or the same applications may tentatively be deployed onto various flexible architectures. Therefore, early performance estimation of the pairing, under imposed real-time constraints, is highly desirable. Such analysis requires a model of both the application and the architecture, and effective means to define the mapping of applicative functions onto architecture resources and services.

AADL and MARTE are two such modeling frameworks. 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 recent OMG UML profile. Despite their many similar features (and MARTE being more detailed on the modeling aspects), AADL provides specific communication schemes between tasks, that need to be represented in MARTE: AADL tasks may be periodic or aperiodic, and in the former case of harmonic or independent periods; communication between tasks may use event-data or pure-data ports (with events triggering the recipient task behavior, while pure data are only sampled and used as such whenever the consuming tasks is otherwise activated).

Representing all these kinds of communications (periodic vs. aperiodic, event-triggered vs. sampled data) in MARTE is not only a challenge, but also an opportunity to provide timed semantics inside the modeling framework (and not aside, as separately provided semantic interpretation to time attributes). We build this semantic construction using MARTE Time Model [3], which is intended exactly for this: specifying in a formal fashion new timed domains of computation and communication.

We exemplify this approach by dealing with the same example as used in AADL [4] to explain the computation of end-to-end flow latencies (and various other related features) in a case of three threads with various rates (periodic or not) and connected through event-data or pure-data links. We show how these formulas are derived from time relations that are integral parts of the MARTE model.

2 A brief AADL overview

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.

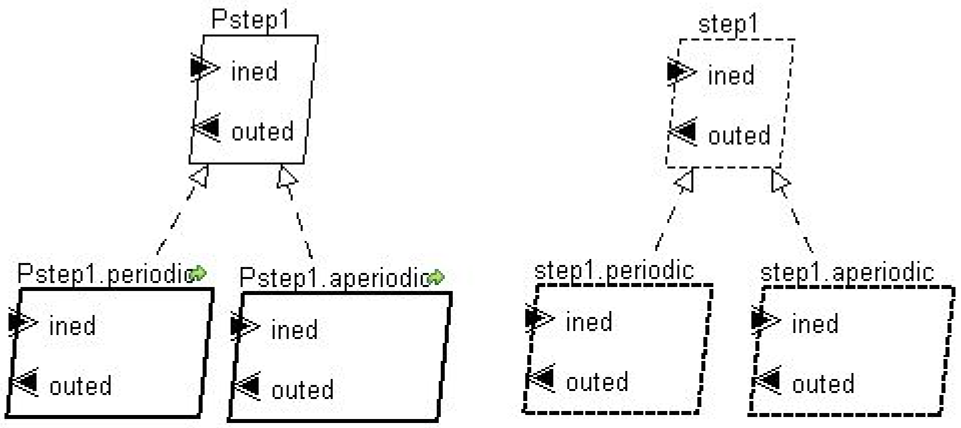

Figure 1. Declaration and implementation of both a process and a thread.

Figure 1 presents the declarations of one process type and one thread type. Each of these declarations comes with the declaration of two possible implementations, one periodic and one aperiodic. Since threads are executed within the context of a process, the process implementations must specify the number of threads it executes and their interconnections. Figure 2 illustrates the case where a process executes two threads (t1 and t2) sequentially.

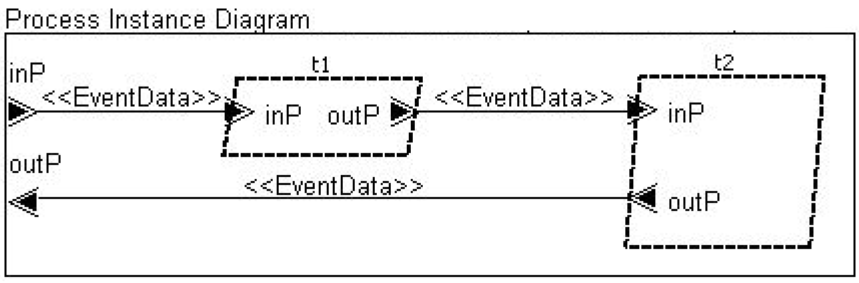

Figure 2. Two threads executed sequentially.

Type and implementation declarations also provide a set of properties to characterize model elements. For threads, the 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.
AADL end-to-end flows explicitly identify a data-stream from sensors to the external environment (actuators). Figure 3 represents such a flow. Note that this model is really an excerpt of the complete model and implies that the threads are declared within the context of one or many processes and bound to an execution host, i.e., a processor. All this contextual information is absolutely required to make the latency analysis since the execution platform and the topology determines the actual parallelism available. When executing on a single processor platform, the threads have to be serialized whereas a dual processor platform offers more parallelism. Figure 8 gives a more faithful view of the complete model.

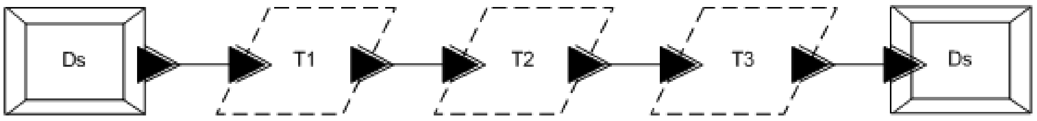

Figure 3. Flow extracted from an AADL model.

There are three kinds of ports in AADL: data, event and event-data. Data ports are for data transmissions without queuing. Connections between data ports are either immediate or delayed. Event ports are for communications of events that may be queued. The size of the queue may induce transfer delays that must be taken into account when performing latency analysis. Event data ports are for message transmission with queuing, here again the queue size may induce transfer delays. On Figure 3, all components have event-data ports represented as solid triangles (as for data ports) with their shadows (as for event ports).

3 MARTE for AADL

The emerging UML2 Profile for MARTE is expected to be the basis for UML representation of AADL models [5]. The adopted MARTE OMG specification provides guidelines in this direction. The main goal of this paper is to further investigate how specific AADL concepts required for end-to-end flow latency analysis can be represented in MARTE. As such, this work is not (yet?) included in the official standard annex.
The idea here is to define, once and for all, a model library for AADL with MARTE. The end-user is not expected to enter into each and every detail about this library and most of the time he should be able to use it as a black box. The following section illustrates the use of this li-

brary on two selected examples. For brevity we only present here the model elements required for dealing with the latency analysis example.

3.1 AADL application software components with MARTE.

The first step is to create classifiers to represent AADL threads. We use the stereotype SwSchedulableResource from the Software Resource Modeling sub-profile (see Figure 4). Only classifiers for periodic and aperiodic threads are shown here.

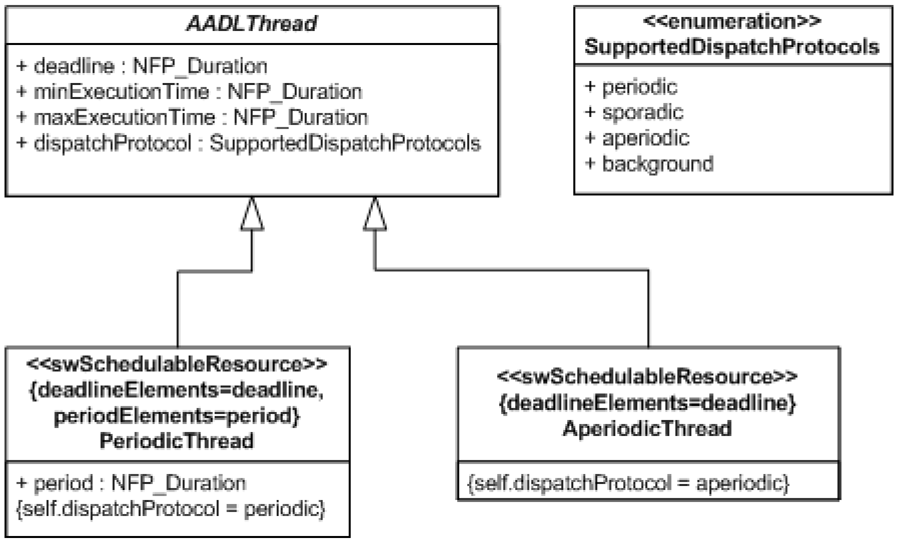
AADLThread
+ deadline : NFP_Duration
+ minExecutionTime : NFP_Duration
+ maxExecutionTime : NFP_Duration
+ dispatchProtocol : SupportedDispatchProtocols
<<enumeration>>
SupportedDispatchProtocols
+ periodic
+ sporadic
+ aperiodic
+ background
<<swSchedulableResource>>
{deadlineElements=deadline, periodElements=period}
PeriodicThread
+ period : NFP_Duration
{self.dispatchProtocol = periodic}
<<swSchedulableResource>>
{deadlineElements=deadline}
AperiodicThread
{self.dispatchProtocol = aperiodic}

Figure 4. AADL thread declarations.

The properties deadlineElements and periodElements are explicitly identified so as to help model transformation tools to extract the right property. This makes it easier for the transformation tools to be language and domain independent. Hence, the exact spelling of the property names does not matter as long as they are referenced by the stereotype properties. Note that contrary to AADL only periodic threads have a property called period. The MARTE equivalent to the AADL type Time is NFP_Duration, defined in the MARTE::BasicNFP_Types (Non Functional Property Types) model library. An NFP_Duration value is defined as a tuple containing a real value and a time unit, among others. Figure 5 shows examples of thread instances, one being periodic and the other one being aperiodic.

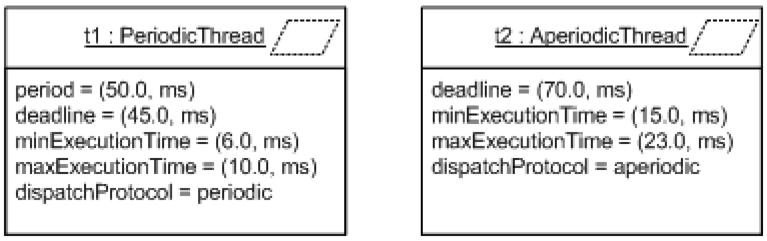
t1 : PeriodicThread
period = (50.0, ms)
deadline = (45.0, ms)
minExecutionTime = (6.0, ms)
maxExecutionTime = (10.0, ms)
dispatchProtocol = periodic
t2 : AperiodicThread
deadline = (70.0, ms)
minExecutionTime = (15.0, ms)
maxExecutionTime = (23.0, ms)
dispatchProtocol = aperiodic

Figure 5. Two thread instances.

NFP_Duration only supports the description of duration values associated with an ideal chronometric clock, which is exactly what AADL supports. Had we wanted to support logical clocks we would have used the templateable TimedValueType defined in the MARTE::Time model library.

3.2 AADL hardware components with MARTE.

The Hardware Resource Modeling subprofile is used to model AADL hardware components (bus, memory, processor, and device). Possible equivalents using MARTE are given in Figure 6.

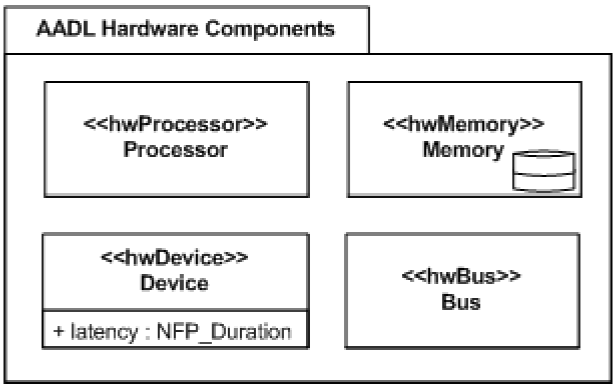

Figure 6. AADL hardware components.

Actually specific classifiers should be customized depending on the physical characteristics of the hardware components (throughput or latency of memories, clock speed of processors, etc.). The latency analysis performed here does not use any specific property, apart from the device latency, so we need not specializing these classifiers further.

3.3 AADL ports with MARTE.

UML component diagrams provide ports and connectors to connect components. The queuing policy should rather be represented on the algorithm itself, i.e., on a UML activity diagram. Activities are composed of actions. The ordering in which the actions are executed is given by a control flow. Data communications between the actions are represented with object flows. By default, an object flow has a queue, the size of which can be parameterized with its property upperBound. So object flows can be used to represent AADL communications using either event or event-data ports. UML allows the specification of a customized selection policy to select which one of the tokens stored in the object node is selected. Unfortunately, the selection behavior must select only one token making it impossible to represent the AADL dequeue protocol AllItems. This protocol dequeues all items from the port every time the port is read. Therefore, only the dequeue protocol OneItem is supported.
To model data ports, UML provides DataStore nodes. On these nodes, the tokens are never consumed thus allowing for multiple readings of the same token. Using a data store node with an upper bound equal to one is a good way to represent communications through data ports.
On Figure 7, the upper part shows an event-based communication with a queue size of 4. The lower part illustrates a data-based communication.

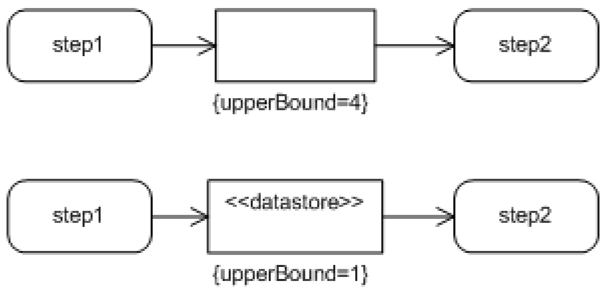

Figure 7. Event or data communications.

The difference between immediate and delayed communications is addressed in the next sub section, since it is not really a structural matter but rather a temporal aspect.

3.4 AADL MoCC with MARTE.

Aside the model elements, the time semantics of these elements must be defined. On one hand, the model of computation, i.e., when the processing starts, finishes or is aborted. On the other hand, the model of communications, i.e., what kind of communication is used. The MARTE Time subprofile, inspired from the theory of tags systems [6], provides a set of general mechanisms to define MoCC. These modeling aspects should be hidden to end-users and we show here how to use MARTE, as a model architect, to build a partial MoCC suitable for AADL. To specify the time constraints with MARTE we use the stereotype ClockConstraint that extends the metaclass UML::Constraint. The language to be used on these clock constraints is called Clock Constraint Specification Language (CCSL) and is defined as an annex of the MARTE specification.

Overall two kinds of communications are possible with AADL. Data-driven communications, where the execution of a given task is triggered by the availability of the data produced by the preceding (according to the order defined by the control flow) task(s). Sampled communications, where pure data are only sampled and used as such whenever the task is otherwise activated. Note that the nature (event, event-data, or data) of the ports involved in the communication is not enough to determine its kind.

For instance, a data-driven communications exist in chains of aperiodic tasks (devices or threads) connected by event or event-data ports. They also exist with synchronous periodic tasks connected by data-ports through an immediate connection. In that latter case, the consuming task becomes aperiodic and its execution is triggered by the completion of the producing task. The CCSL clock relation alternatesWith models data-driven communications.

step1.finish **alternatesWith** step2.start (1)

Eq. 1 illustrates a data-driven communication from step1 to step2. Note that this constraint is not symmetrical since the completion of step1 may cause the execution of step2, but not the converse.

Sampled communications occurs in various cases. For instance, when two asynchronous tasks (whether periodic or not) communicate, but also when two synchronous periodic tasks are connected by data-ports through a delayed connection.

step2.start ≡ step1.finish **sampledTo** ^step2 (2)

Eq. 2 illustrates a sampled communication from step1 to step2. ^step2 represents the activation condition of step2. If step2 is a periodic thread, its activation condition can be defined using the CCSL relation isPeriodicOn (see Eq. 3–4).

c100 ≡ idealClk **discretizedBy** 0.01 (3)
^step2 **isPeriodicOn** c100 **period**=10 (4)

idealClk is defined in the MARTE Time library and stands for a dense chronometric (related to physical time) perfect (with no jitter or any other flaw) clock. Eq. 3 defines c100 by discretizing idealClk. The default unit of idealClk is the second (s), so c100 is a 100-hz discrete chronometric clock. ^step2 is periodic on c100 with period 10 (Eq. 4), that makes ^step2 a 10-hz discrete chronometric clock.
Had step2 been an aperiodic thread, its activation condition would have been an unbound logical clock.

4 An AADL example

In this section we combine all these elements into a complete model that derives from [4]. It is displayed using the OSATE Eclipse plug-in environment for AADL [7] in Figure 8.

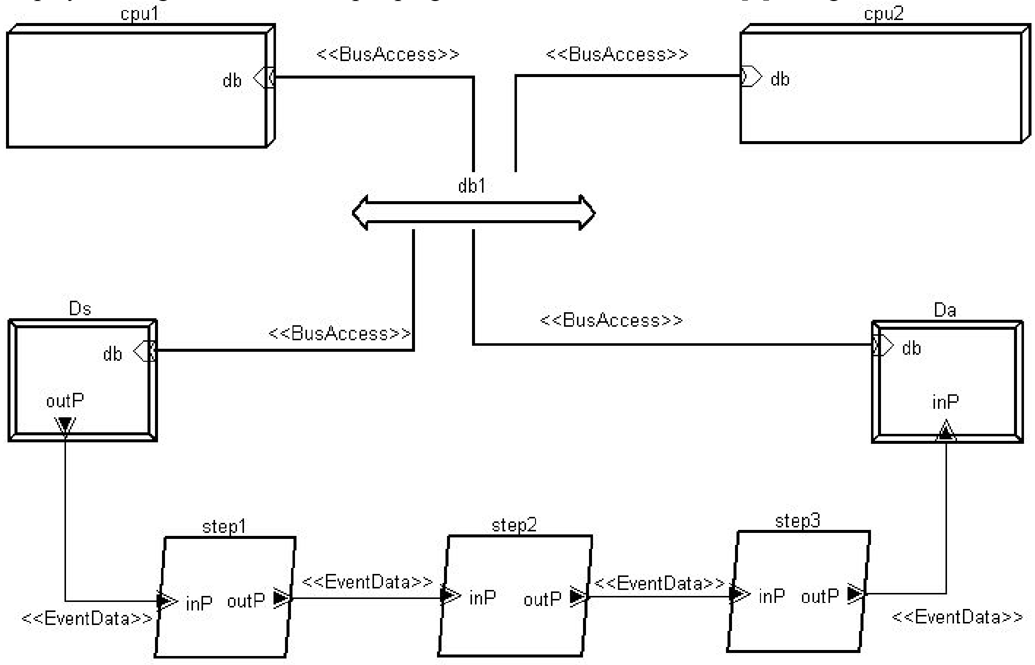

Figure 8. The System in OSATE

This represented flow starts from a sensor (an aperiodic device instance) and sinks in an actuator (also aperiodic) through three process instances. Each process executes a single thread. 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. The actual binding is not represented on this figure as we have ignored the effects of communications, just as in the original example [4]. The component declarations and implementations are not presented here. The full AADL code is available in [4].

5 The MARTE representation

5.1 The fully aperiodic case

The simplest case is when all threads are aperiodic (see Figure 9) and therefore all communications are data-driven.

We start by describing the model algorithm with an UML activity diagram (see Figure 9, uppermost part). All communications are through event-data ports with infinite queues. Two actions (acquire and release) have been added to represent the behavior of the two devices, compare with Figure 8.
AADL software (Figure 9, middle part) and execution platform (Figure 9, lower part) components are modeled with composite structure diagrams using the classifiers defined in Section 3. A fully asynchronous application requires no additional MARTE constraints since that the default semantics of UML activity diagram. In following cases we use MARTE to modify the semantics of computation nodes and communications.

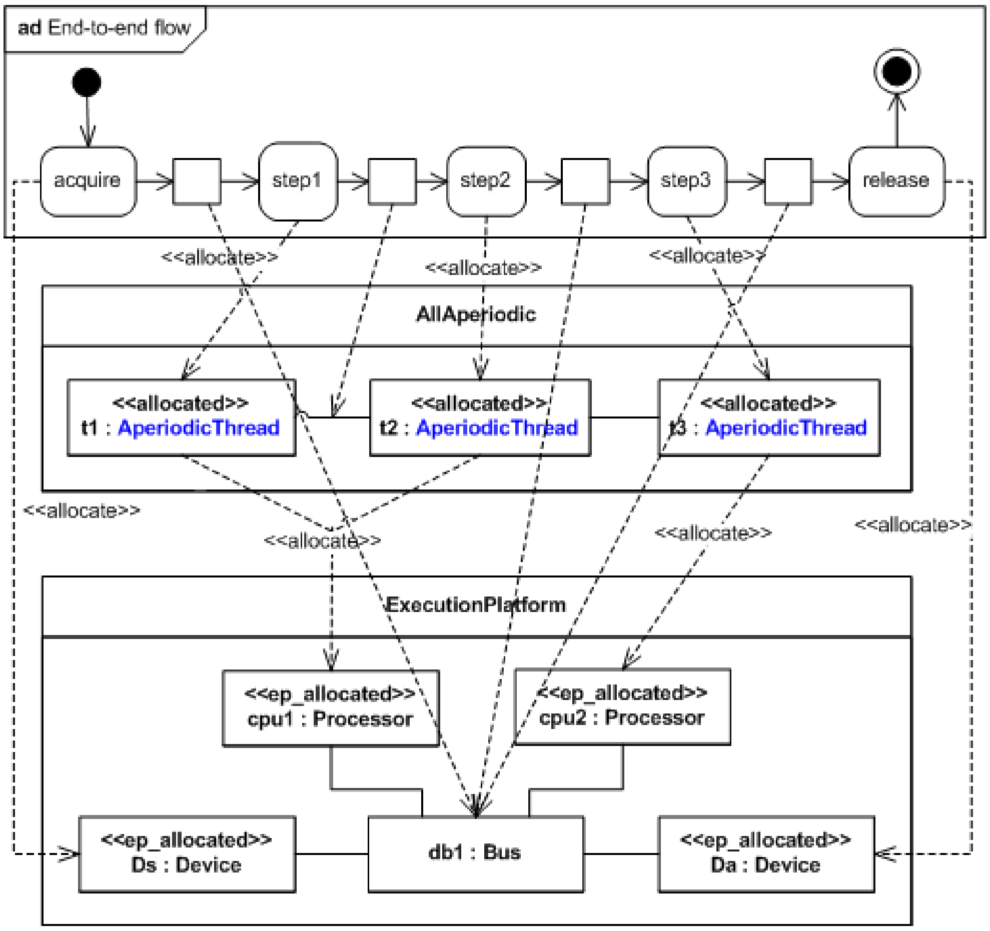
ad End-to-end flow
acquire
step1
step2
step3
release
<<allocate>>
<<allocate>>
<<allocate>>
AllAperiodic
<<allocated>> t1 : AperiodicThread
<<allocated>> t2 : AperiodicThread
<<allocated>> t3 : AperiodicThread
<<allocate>>
<<allocate>>
<<allocate>>
<<allocate>>
ExecutionPlatform
<<ep_allocated>> cpu1 : Processor
<<ep_allocated>> cpu2 : Processor
<<ep_allocated>> Ds : Device
db1 : Bus
<<ep_allocated>> Da : Device

Figure 9. MARTE model, all-aperiodic case.

The AADL binding mechanism finds its equivalent in the MARTE allocation package. First, actions and object nodes are allocated (dashed arrows on Figure 9) to software components. Second, software components are allocated to execution platform model elements.

All these annotations (stereotypes) can be extracted using model-driven engineering techniques and fed into time analysis tools, including AADL latency analysis tool. Then, we go a bit further than AADL, by bringing back the latency analysis results into UML and MARTE in the form of timing diagrams (Figure 10). The timing diagram represents a family of possible schedules for a given execution flow and a given pair application/execution platform.

The intervals on the duration constraints reflect AADL semantics of tasks. The computation execution time of devices is their latency whereas for threads it ranges between the minimum execution time (MET_{ti}) and their deadline (D_{ti}).

Bold horizontal lines represent the execution of threads whereas in UML they are supposed to represent different states. The vertical plain connectors between the horizontal lines represent the communication between tasks, which is assumed here to be instantaneous.

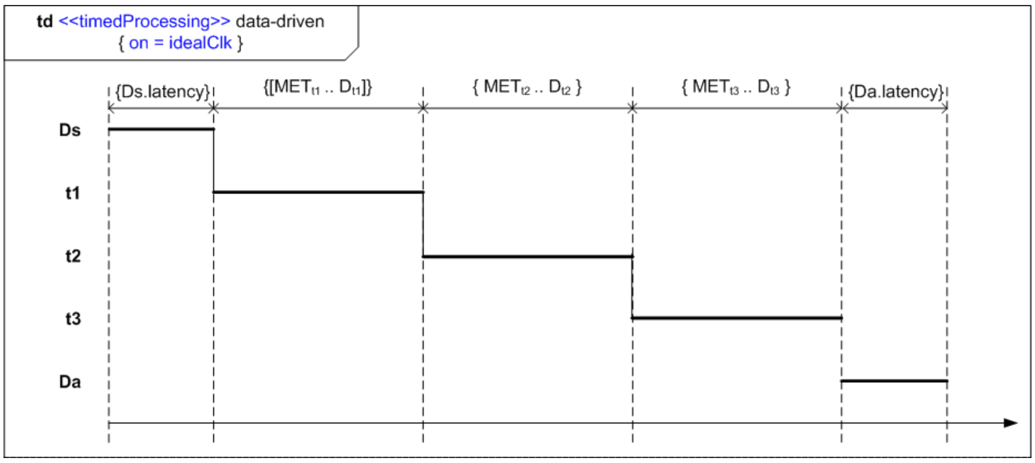

Figure 10. Timing diagram, all-aperiodic case.

The flow latencies both in the worst-case and the best-case can be directly read on this timing diagram, they could also be automatically extracted by a transformation tool. Extracting this information we get the formulas below that exactly match the ones computed by the AADL analysis tool [4].

$$\text{End-to-End Flow Latency} = \text{Ds.latency} + \text{flow latency} + \text{Da.latency}$$

$$\text{Flow Latency}_{\text{worst-case}} = \text{t1.deadline} + \text{t2.deadline} + \text{t3.deadline}$$

$$\text{Flow Latency}_{\text{best-case}} = \text{t1.MinExecTime} + \text{t2.MinExecTime} + \text{t3.MinExecTime}$$

$$\text{Latency Jitter} = \Sigma_i(\text{ti.deadline} - \text{ti.MinExecTime})$$

In the fully asynchronous case, the latency jitter is maximal and the system less predictable. That is one reason to put barrier synchronization mechanisms so to increase predictability.

5.2 The fully periodic case

Figure 11 represents the case where all threads are periodic.

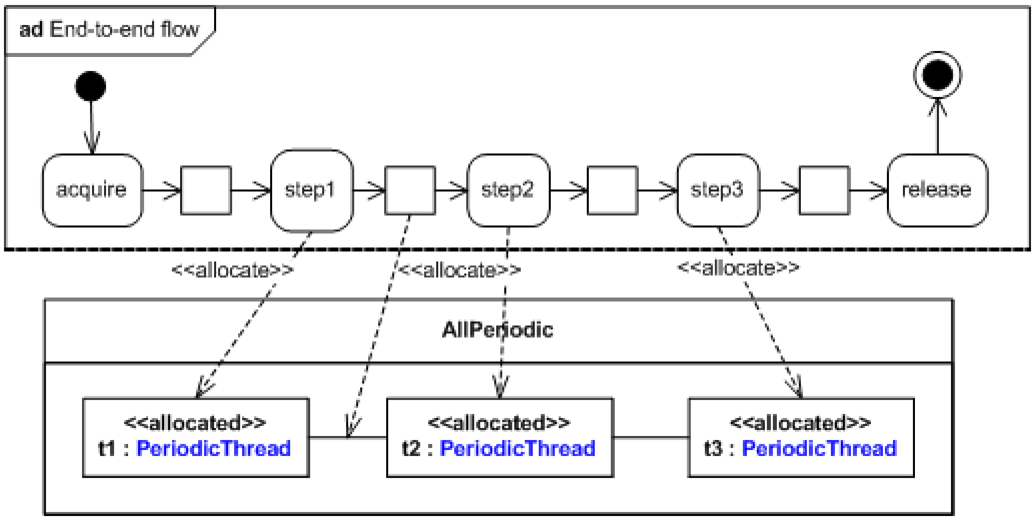

Figure 11. MARTE model, all periodic case.

Following the methodology defined in Section 3.4., we can define adequate clock constraints to modify the UML Activities semantics so that they behave as in AADL. We can consider several cases depending on the relations among the three periodic threads.

5.2.1 The synchronous case.

When all threads have the same period P and if their dispatches are aligned, they are synchronous.

$$\text{clk} \equiv \text{idealClk } \textbf{discretizedBy } \text{P} \tag{5}$$

Eq. 5 declares the common clock to all these periodic threads. Figure 12 represents the CCSL constraints and an equivalent graphical representation. Green arrows represent the communications and Dashed arrows the instant relations. A plain blue arrow denotes strict precedence whereas an empty blue arrow denotes non-strict precedence. The first two lines can then be interpreted as follows.

The instant at which the action acquire (Ds) finishes (Ds.finish) is located between two ticks of the clock, let clk_i be the earlier of these instants (Eq. 6).

$$(\square\exists i \in N^*)\ (\text{Ds.finish} \in]\text{clk}_i, \text{clk}_{i+1}]) \tag{6}$$

This implies that the action step1 must follow the tick clk_{i+1} (Eq. 7).

$$\text{step1.start} \in [\text{clk}_{i+1}, \text{clk}_{i+2}[) \tag{7}$$

That is the definition of an **asynchronous** sampling. The data emitted by the asynchronous device Ds is sampled by the synchronous thread t1, according to its clock clk.

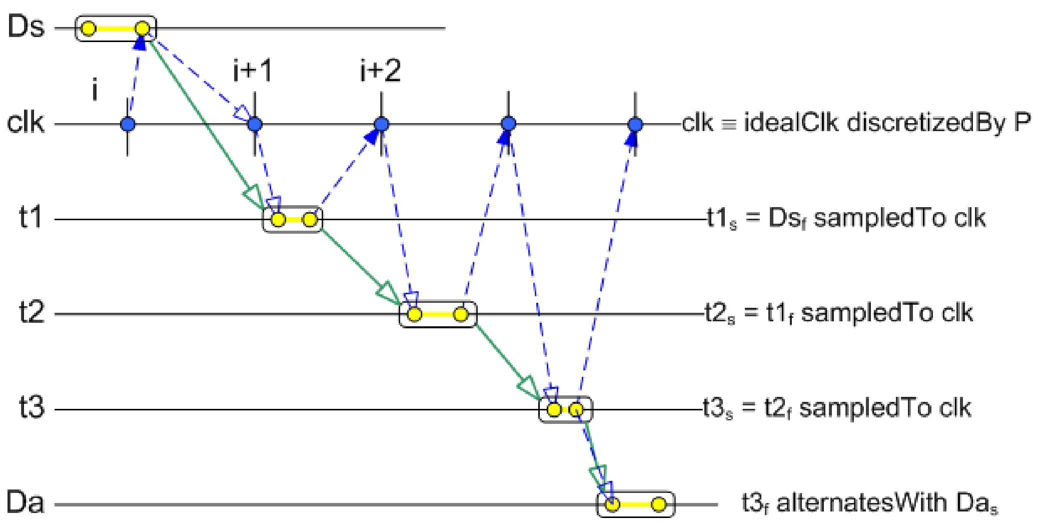

Figure 12. CCSL constraints, the synchronous case.

Since all threads are periodic and synchronous, the three inter-thread communications (from acquire to step1, from step1 to step2, and from step2 to step3) are sampled communications. The last communication (from step3 to release) is data-driven, since the device is aperiodic.

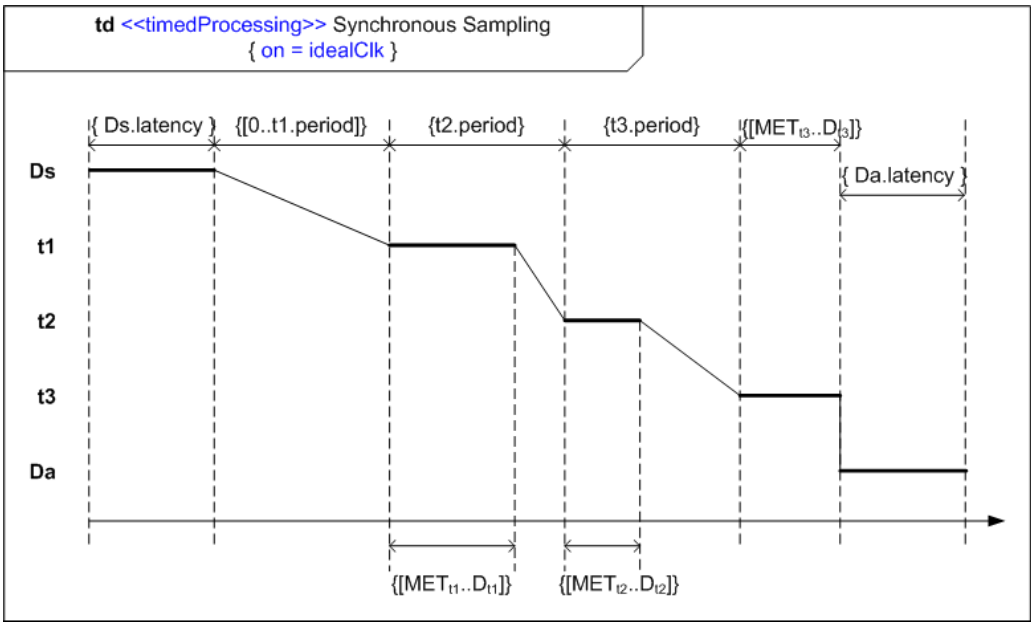

Figure 13. Timing diagram, the synchronous case.

Oblique lines linking two computation lines represent the communications between two tasks and the sampling delays. For sampled communications, this amounts to wait for the next tick of the receiver clock. The maximal sampling delay is when the communication waits for the full period because the previous tick has just been missed. It is not normative in UML timing diagrams to have these “oblique” lines, but it is a convenient notation to represent intermediate communication states between two steady processing states (e.g., between Ds and t1). Assuming, as in [4], that the sampling delays are always maximal, we get the same formulas (reproduced below) as the AADL latency analysis tool. We could also have derive more optimistic scenarios where the sampling delays are not necessarily worst cases.

End-to-End Flow Latency = Ds.latency + flow latency + Da.latency

$\text{Flow Latency}_{\text{worst-case}}$ = t1.period + t2.period + t3.period + t3.deadline

$\text{Flow Latency}_{\text{Best-Case}}$ = t1.period + t2.period + t3.period + t3.MinExecTime

Latency jitter = t3.deadline – t3.MinExecTime

5.2.2 The asynchronous case

When all the threads are periodic but are asynchronous, i.e. they all have different dispatch conditions (clocks) and there are no *a priori* relations between these clocks, the situation is much more complex.

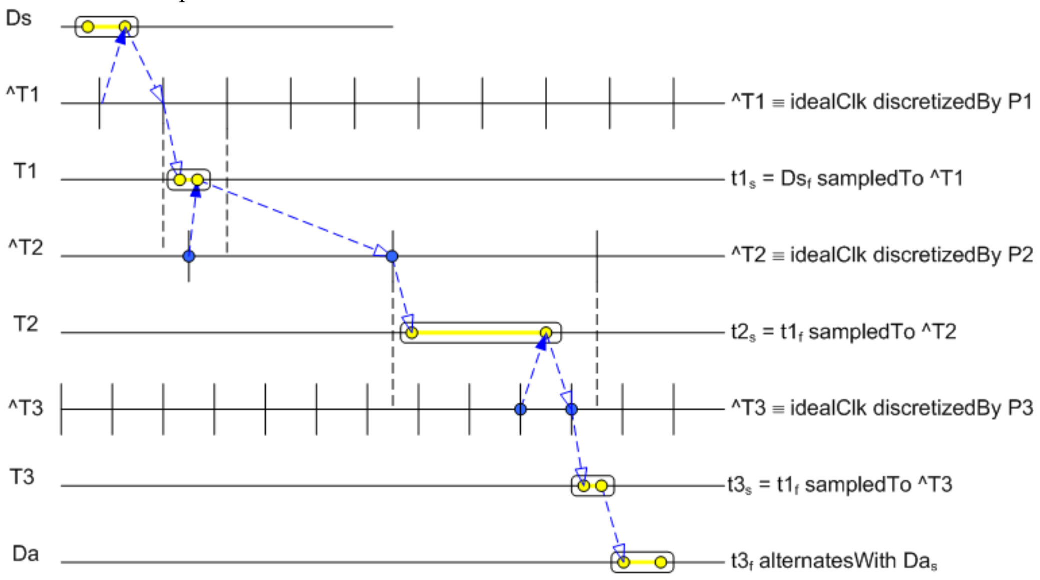
Ds
^T1
^T1 ≡ idealClk discretizedBy P1
T1
$t1_s = Ds_f$ sampledTo ^T1
^T2
^T2 ≡ idealClk discretizedBy P2
T2
$t2_s = t1_f$ sampledTo ^T2
^T3
^T3 ≡ idealClk discretizedBy P3
T3
$t3_s = t1_f$ sampledTo ^T3
Da
$t3_f$ alternatesWith Da_s

Figure 14. CCSL constraints, the asynchronous case.

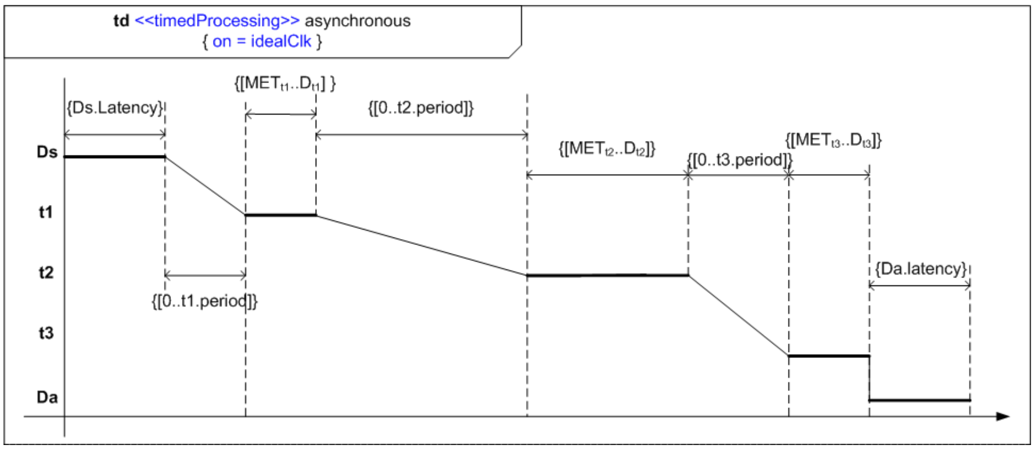

Figure 15. Timing diagrams, the asynchronous case.

Not only the latency jitter is maximal but both the worst-case and best-case flow latencies are more pessimistic.

End-to-End Flow Latency = Ds.latency + flow latency + Da.latency

Flow latency$_{\text{worst-case}}$ = Σ_i(ti.period + ti.deadline)

Flow Latency$_{\text{best-case}}$ = Σ_i(ti.period + ti.MinExecTime)

Latency Jitter = Σ_i(ti.deadline – ti.MinExecTime)

5.2.3 The harmonic down-sampling case

When all threads are not fully synchronous but the dispatches are aligned, we get better results. In this example, t1 and t3 are synchronous with period P but t2 is twice slower and the dispatches are still aligned (every other instant). We have harmonic periods between t1 and t2.

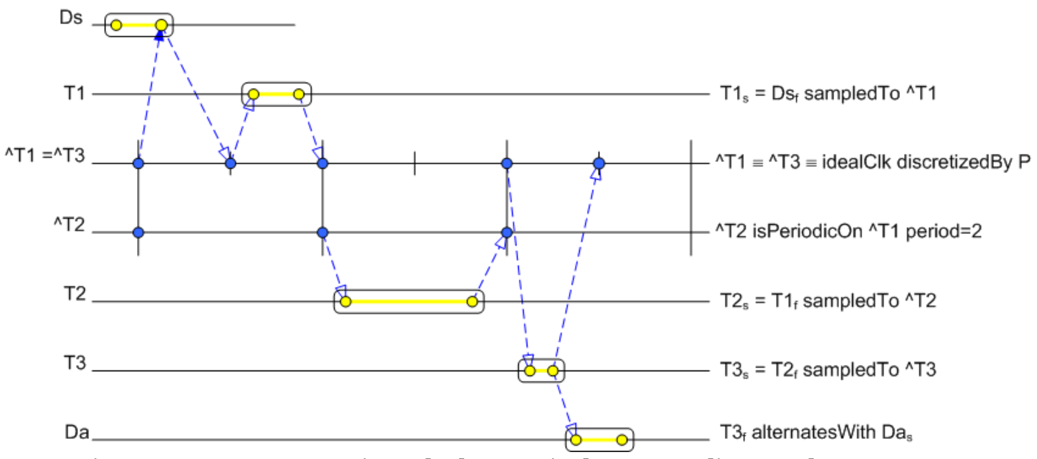
Ds
T1
^T1 =^T3
^T2
T2
T3
Da
$T1_s$ = Ds_f sampledTo ^T1
^T1 ≡ ^T3 ≡ idealClk discretizedBy P
^T2 isPeriodicOn ^T1 period=2
$T2_s$ = $T1_f$ sampledTo ^T2
$T3_s$ = $T2_f$ sampledTo ^T3
$T3_f$ alternatesWith Da_s

Figure 16. CCSL constraints, the harmonic down-sampling synchronous case.

^T1 ≡ ^T3 ≡ idealClk **discretizedBy** P (8)

^T2 **isPeriodicOn** ^T1 **period=**2 (9)

Eq. 8 defines the common clock to t1 and t3 by discretizing idealClk. Eq. 9 uses the isPeriodicOn constraint to define the dispatch condition of t2 according to ^T1.

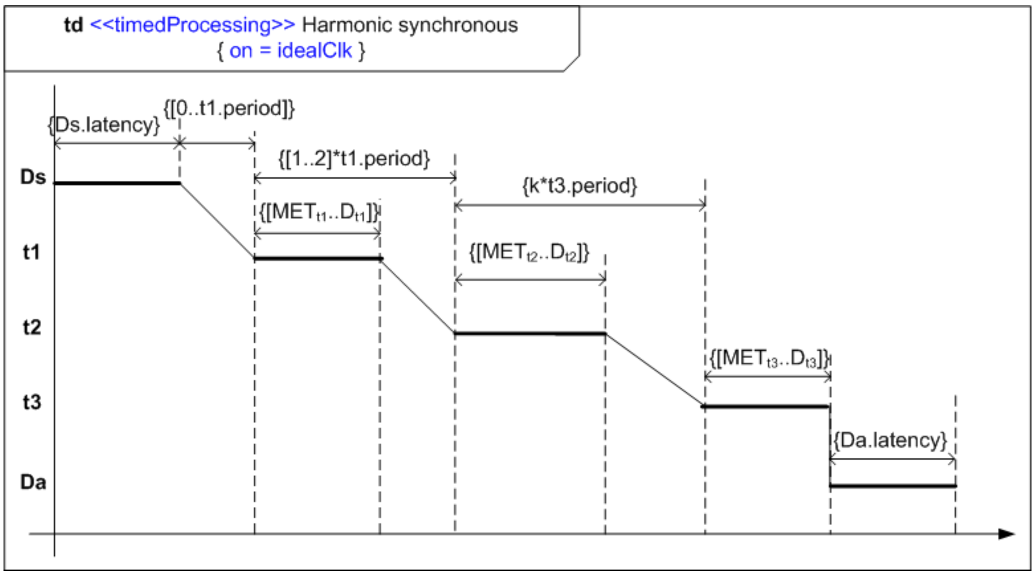

Figure 17. Timing diagrams, the harmonic synchronous case.

In the equations below, k1 is either 1 or 2 since f1/f2 = 2.

End-to-End Flow Latency = Ds.latency + flow latency + Da.latency

Flow latency$_{\text{worst-case}}$ = t1.period + k1*t1.period + k2*t3.period + t3.deadline

Flow Latency$_{\text{best-case}}$ = t1.period + k1*t1.period + k3*t3.period + t3.MinExecTime

Latency jitter = (k2-k3)*t3.period + t3.deadline – t3.MinExecTime

5.3 The mixed periodic-aperiodic case

We study here two possible configurations extracted from [4] where periodic and aperiodic threads are mixed.

5.3.1 Aperiodic-Periodic-Aperiodic

In this mixed periodic-aperiodic case, threads t1 and t3 are aperiodic whereas thread t2 is periodic (Figure 18).

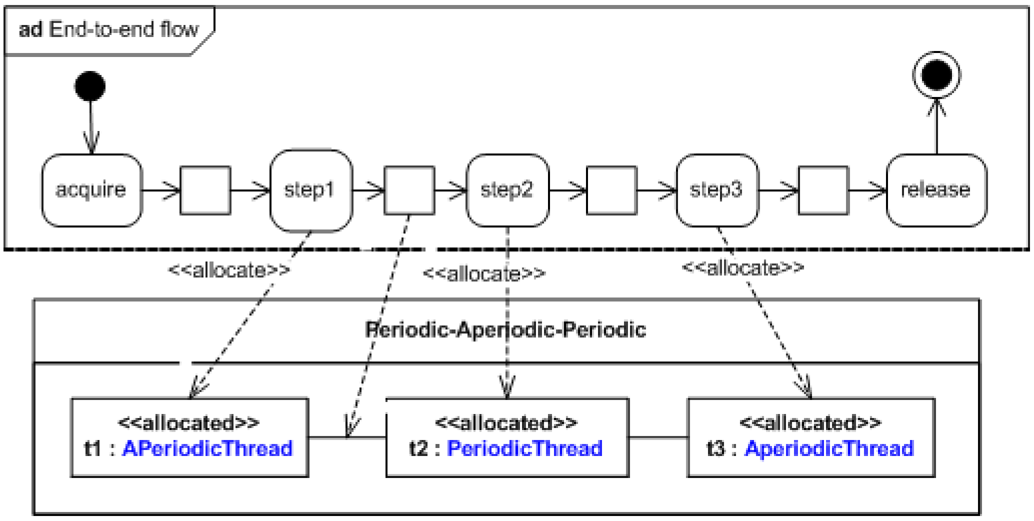

Figure 18. MARTE model, Aperiodic-Periodic-Aperiodic case.

The adequate CCSL constraints are given in Figure 19 and the deduced timing diagram in Figure 20). The flow latency formulas follow.

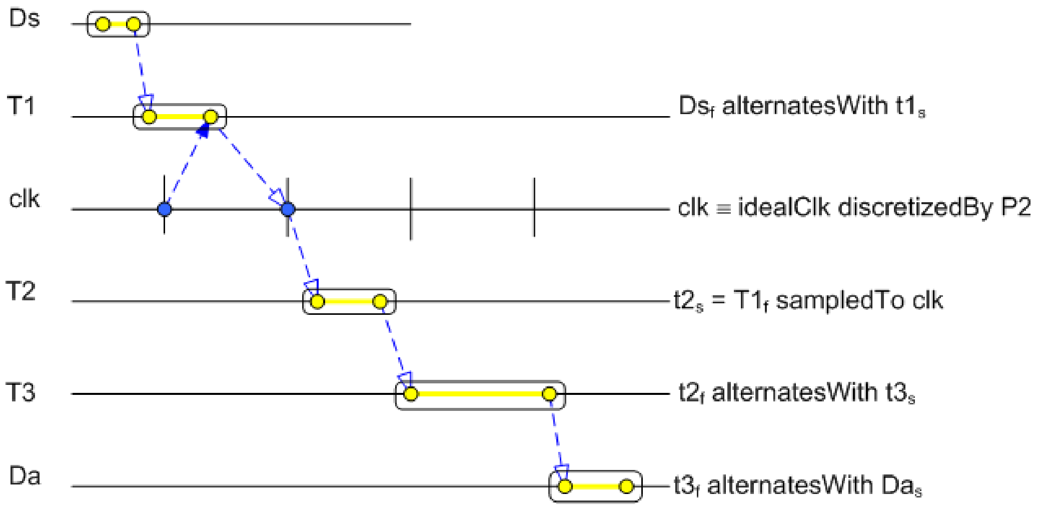

Figure 19. CCSL constraints, Aperiodic-Periodic-Aperiodic case.

End-to-End Flow Latency = Ds.latency + flow latency + Da.latency

Flow Latency$_{\text{wrost-case}}$ =t1.deadline + t2.period + t2.deadline + t3.deadline

Flow Latency$_{\text{best-case}}$ =t1.MinExecTime + t2.period + t2.MinExecTime + t3.MinExecTime

Latency Jitter = Σ_i(ti.deadline – ti.MinExecTime)

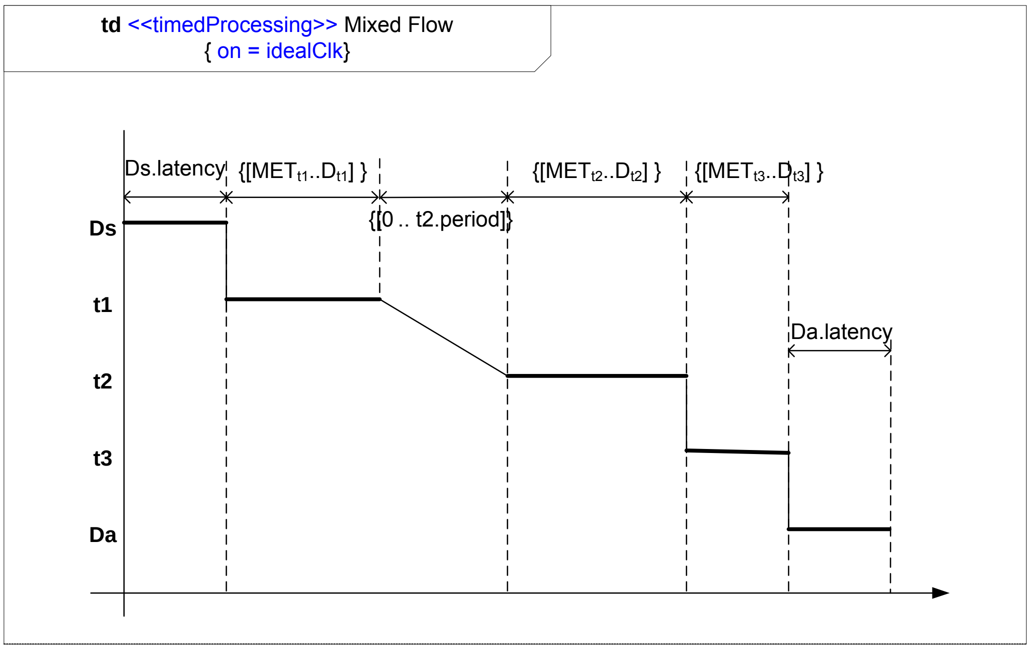
td <<timedProcessing>> Mixed Flow
{ on = idealClk}
Ds.latency
{[MET$_{t1}$..D$_{t1}$] }
{[0 .. t2.period]}
{[MET$_{t2}$..D$_{t2}$] }
{[MET$_{t3}$..D$_{t3}$] }
Da.latency
Ds
t1
t2
t3
Da

Figure 20. Timing diagram, Aperiodic-Periodic-Aperiodic case.

5.3.2 Periodic-Aperiodic-Periodic

This case only differs from the all-periodic case by making aperiodic the thread t2 (Figure 21).

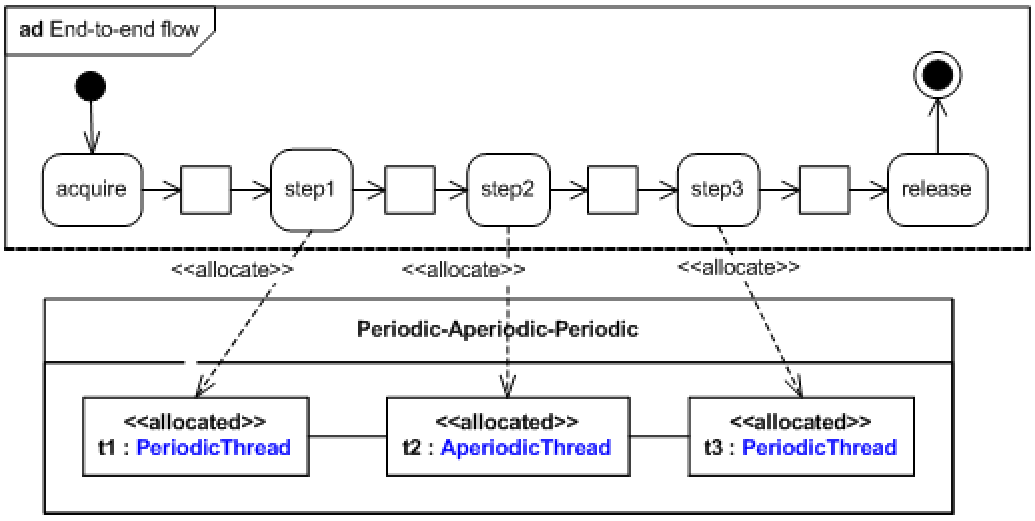

Figure 21. MARTE model, periodic-aperiodic-periodic case.

In this configuration, the communication from step1 to step2 becomes data-driven. The CCSL constraint is adapted accordingly (Figure 22). We also get a different timing diagram (Figure 23) and different flow latency formulas.

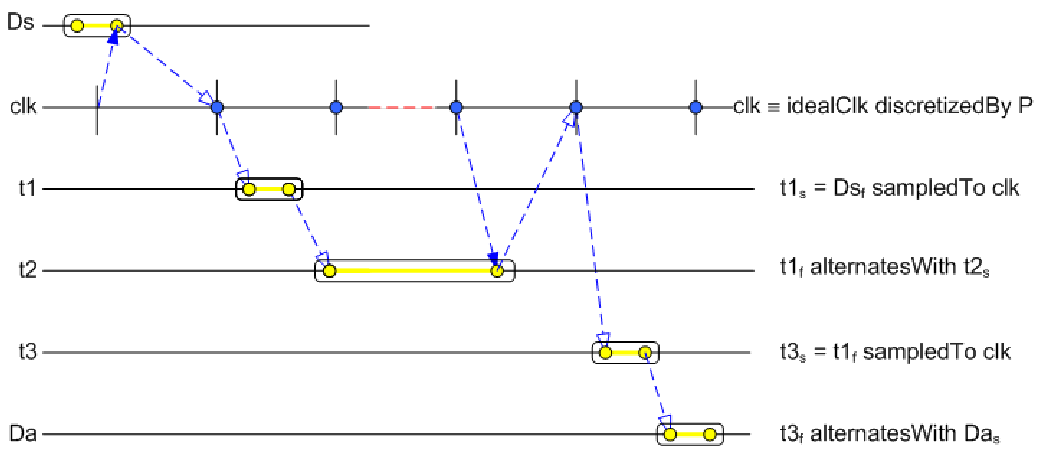

Figure 22. CCSL constraints, mixed case.

The constants k1 and k2 in the formulas below come from the synchronization on clk of the result emitted by t2. The computation execution time of the thread t2 is rounded up to the next multiple of t3.period. It need not be the same multiple in the best-case (k2 using the minimum execution times) or in the worst-case (k1 using the deadlines).

Flow latency$_{\text{Worst-Case}}$ = t1.period + k1 * t3.period + t3.deadline

k1 = $\lceil$ (t1.deadline + t2.deadline) / t3.period $\rceil$

Flow Latency$_{\text{Best-Case}}$ = t1.period + k2 * t3.period + t3.MinExecTime (k2$\leq$k1)

k2 = $\lceil$ (t1.MinExecTime + t2.MinExecTime) / t3.period $\rceil$

Latency jitter = t3.deadline – t3.MinExecTime + (k1-k2)*t3.period

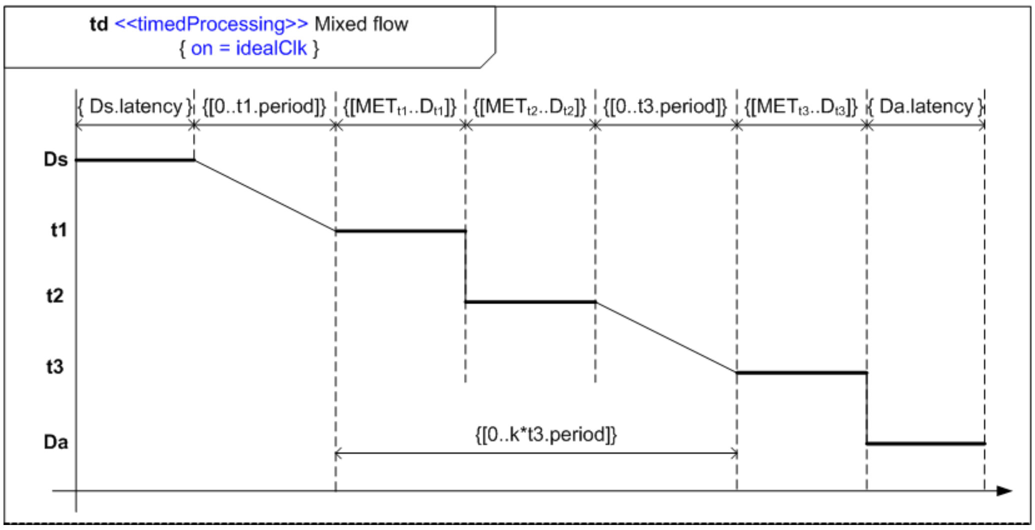

Figure 23. Timing diagram, periodic-aperiodic-periodic case.

6 Conclusion

We have shown how MARTE could be used to model mixed systems with both periodic and aperiodic tasks, which is a big issue while modeling embedded systems. We have compared

MARTE and AADL in this matter, highlighting MARTE capabilities to make the computation formulas explicit. Several different configurations involving event-data ports are illustrated. Few other cases involving data ports are studied in [8].

More generally, MARTE and its time model could be used to model various timed models of computation and communication.

7 Bibliography

[1] SAE: Architecture Analysis and Design Language (AADL). June 2006, document AS5506/1. http://www.sae.org/technical/standards/AS5506/1.

[2] OMG: UML profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE), beta 1, August 2007, document ptc/07-08-04.

[3] André C, Mallet F, de Simone R (2007): Modeling Time(s). Springer LNCS 4735:559-573.

[4] Feiler P.H, Hansson J: Flow Latency Analysis with the Architecture Analysis and Design Language (AADL). Carnegie Mellon University, Technical Note CMU/SEI-2007-TN-010, June 2007.

[5] Faugère M, Bourbeau T, de Simone R, Gérard S (2007): MARTE: Also an UML Profile for Modeling AADL Applications. ICECCS:359-364.

[6] Lee E.A, Sangiovanni-Vincentelli A.L (1998): A framework for comparing models of computation. IEEE Trans. On Computer-Aided Design of Integrated Circuits and Systems, 17(12):1217-1229.

[7] OSATE release 1.5.3. http://www.aadl.info

[8] André C, Mallet F, de Simone R (2007): Modeling of Immediate vs. Delayed Data Communications: from AADL to UML MARTE". ECSI FDL 2007.