Software Engineering Institute

Flow Latency Analysis with the Architecture Analysis and Design Language (AADL)

Peter Feiler
Jőrgen Hansson

June 2007

TECHNICAL NOTE
CMU/SEI-2007-TN-010

Performance-Critical Systems

Carnegie Mellon

This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

List of Figures

List of Tables

Abstract

Control system components are sensitive to the end-to-end latency and age of signal data as well as their variation (jitter). Typically, components assume data latency, age, and variation to be within a given range. Different runtime configurations (i.e., different sampling and data driven signal processing pipelines, different communication mechanisms, partitioned architectures, and globally synchronous versus asynchronous hardware) affect these measures.

This technical note describes the flow specification notation in the Architecture Analysis and Design Language (AADL) and introduces a latency analysis framework. This framework is able to determine the end-to-end latency and age of a signal stream data as well as their variation (jitter). The latency calculations are illustrated in the context of an example AADL model. The report illustrates how this latency analysis capability can be used to determine worst-case end-to-end latency on system models of different fidelity and can take into account partitioned architectures. The report summarizes the worst-case end-to-end flow latency analysis capabilities that are provided by the Flow Latency Analysis plug-in for the Open Source AADL Tool Environment (OSATE).

.

1 Introduction

Many embedded systems have control system components. Those control system components process a signal data stream from sensors and affect the external environment (e.g., a physical plant, through actuators). Processing of such a signal stream is time sensitive. The degree of time sensitivity depends on the lag of the physical systems and the responsiveness of the control algorithm. It is also affected by the sampling age of the data (i.e., the amount of time expired since the data was read by a sensor and an output computed with this data is passed to an actuator).

Control algorithms are designed to accommodate for this delay. However, control algorithms are sensitive to variation (jitter) in this delay. For example, Cervin, Årzén,, and Hendrickson [Cervin 2006] illustrate how sampling jitter and end-to-end latency jitter from sensor to actuator affects the stability of controllers. They also show that jitter varies according to the scheduling algorithm for executing a task set. In other words, the choice of runtime system affects end-to-end latency and age as well as their jitter. Their jitter is perceived by the control algorithm as increased noise in the data, which can occur when configuring an embedded application with different scheduling and communication policies or when migrating a proven legacy application to a new platform.

The end-to-end latency and the age of data in a signal stream may differ. End-to-end latency is the amount of time it takes for a new data value from a sensor to get processed and output at the actuator. If data elements are missing or the data stream is oversampled, the same data element may be processed multiple times. In that case, the age of the data output to the actuator may be larger than the end-to-end latency.

There are a number of contributors to the end-to-end latency/age and their jitter, including the following:

- Actual execution time of a task

  The execution time of a task can vary between a minimum execution time and a maximum (or worst-case) execution time. Use of caches in processors may reduce the minimum execution time, but they may not reduce maximum execution time under worst-case assumptions. Preemption by another task may invalidate the cache and result in cache misses.

- Completion time of a task

  The completion time of a task may be later than its worst-case execution time due to other tasks sharing the processor or to synchronization on shared resources. The worst-case completion time of a task in a schedulable system is its deadline. In other words, the presence of other tasks and variation in their execution time affects the completion time of a task.

- Sampling latency

  Tasks may process a data stream in a data-driven manner (i.e., the completion of one task triggers the execution of the next task). In this case, any latency jitter due to tasks is cumulative. Control systems typically use sampling to process the data more deterministically and , manage latency jitter. Sampling occurs at a given rate and is driven by a clock. Sampling increases end-to-end latency.

- Sampling jitter

  Latency jitter may exceed the sampling period. In this case, the sampling task may process the old value sometimes and the new value other times. The mechanism used to communicate data between tasks may also contribute to sampling jitter. For example, communication through a shared data area may result in non-deterministic sampling when tasks execute preemptively on the same processor or concurrently on two different processor cores. For example, a task down-sampling at half the rate may sample two data elements in a row and then skip two data elements instead of sampling every other data element. This results in a sampling variation of two frames.

- Globally asynchronous systems

  In a globally synchronous system, task dispatches are aligned. As a result, the sampling latency can be determined by rounding the computational latency to the next multiple of the sampling rate. In a globally asynchronous system, the sampling latency has to be added to the computational latency to accommodate worst-case assumptions of misalignment of clocks. Furthermore, clock drift adds to latency jitter.

- Partitioned architectures and time-triggered architectures

  Partitioning is used to support integrated modular avionics (IMA). In order to achieve more deterministic behavior, frame-delayed communication is typically used. Frame-delayed communication limits increases in jitter, but it adds to end-to-end latency. Furthermore, frame-delayed communication may double the end-to-end latency in a migration to a partitioned system, when it is combined with an existing task communication mechanism such as periodic I/O through a high-priority task. Similarly, time-triggered architectures operate a deterministic protocol on a system bus to maintain deterministic behavior. Again, this may limit jitter and increase end-to-end latency.

The international, industry standard Architecture Analysis and Design Language (AADL) [SAE AS5506 2004] has the expressive power to model

- signal streams as end-to-end flows
- sampling and data-driven processing as periodic and aperiodic threads that communicate through sampling data ports and queued event data ports
- partitioned and time-triggered architectures

AADL also can map application software onto different hardware platforms and specify ranges of execution times on different platforms, deadlines, and expected latencies along specified data flows. Therefore, AADL models can form the basis for an analytical framework through which we can investigate the impact of the runtime system on end-to-end latency, age, and their jitter and compare those results against the assumptions made by the control algorithms.

In this report, we describe the ability of AADL to determine a lower bound for the worst-case end-to-end latency in a system. If this lower bound value exceeds the desired latency, the requirement is not met. The AADL model may reflect the actual system at different levels of fidelity. As the fidelity increases, the lower bound may increase as well, but it will never decrease. For example, we demonstrate that the end-to-end latency of a signal flow may be determined from a model at the level of subsystems that are mapped into partitions, where those partitions take into

account the latency due to cross-partition communication. The model may be refined to specify latency contributed by an individual subsystem due to processing, or the subsystem may be elaborated into a task model, where execution times, deadlines, and sampling rates are taken into account. The application system may be mapped onto different hardware platforms; in that case, workload on individual processors and communication latency can be taken into account in the end-to-end latency analysis.

This technical note summarizes the flow latency analysis capabilities that are provided by the Flow Latency Analysis plug-in for the Open Source AADL Tool Environment (OSATE). The flow latency analysis capability utilizes the ability of the AADL to support specification of end-to-end flows through a sequence of system components.

In Section 2, we describe the flow specification notation in AADL. In Section 3, we introduce a latency analysis framework for calculating the end-to-end latency, and in Section 4 we discuss its use on system models. In Section 5, we explain how the Flow Latency Analysis plug-in can be used on system models of different fidelity.

2 Flow Specifications and Flow Instances

A flow specification describes an externally observable flow of application logic through a component. Such logical flows may be realized through ports and connections of different data types and a combination of data, event, and event data ports. Flow specifications represent

- flow sources—flows originating from within a component
- flow sinks—flows ending within a component
- flow paths—flows through a component from its incoming ports to its outgoing ports

Flow instances describe actual flow sequences through components and sets of components across one or more connections. They are declared in component implementations. A flow sequence takes one of two forms:

1. A flow implementation describes how a flow specification of a component is realized in its component implementation.
2. An end-to-end flow specifies a flow that starts within one subcomponent and ends within another subcomponent.

Flow specifications, flow implementations, and end-to-end flows can have expected and actual values for flow-related properties (e.g., latency or rounding error accumulation).

The purpose of specifying end-to-end flows is to support various forms of flow analysis, such as end-to-end timing and latency, reliability, numerical error propagation, Quality of Service (QoS), and resource management based on operational flows. To support such analyses, relevant properties are provided for the end-to-end flow, the flow specifications of components, and the ports involved in the flow to be analyzed. For example, to deal with end-to-end latency, the end-to-end flow may have properties specifying its expected maximum latency and actual latency. In addition, ports on individual components may have flow-specific properties (e.g., an **`in`** port property specifies the expected latency of data relative to its sensor sampling time or in terms of end-to-end latency from sensor to actuator to reflect the latency assumption embedded in its extrapolation algorithm).

2.1 SPECIFICATION OF EXTERNALLY VISIBLE FLOWS

A flow specification declaration in a component type specifies an externally visible flow through a component's ports, port groups, or parameters. The flow through a component is called a flow path. A flow originating in a component is called a flow source. A flow ending in a component is called a flow sink. Figure 1 illustrates a system called *GPS* with three ports and two flow specifications. These are the flows through *GPS* and out of *GPS* that are externally visible. The flow path symbol is connected to two ports, while flow source symbol connected to one port.

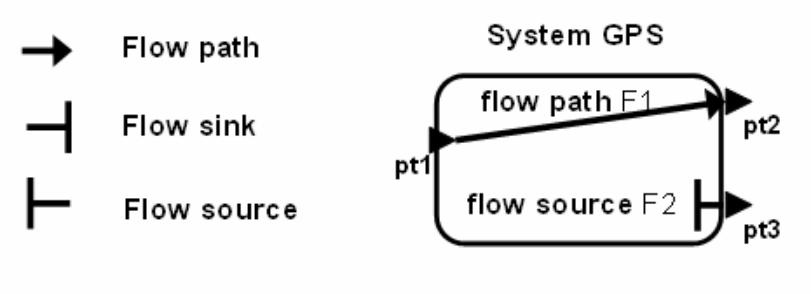

Figure 1: Flow Specifications for GPS System

The ports identified by a flow specification can have different data and port types (i.e., one can be an event port and the other an event data port). Also, multiple flow specifications can be defined involving the same ports. For example, data coming in through an **`in`** port group is processed and data derived from one of the port group's contained ports is sent out through different **`out`** ports. This capability allows logical flows of information through components to be characterized by attributing flow specifications and the ports involved in flow specifications with relevant AADL property values. Properties other than the set of predeclared properties can be introduced through the AADL Property Set concept [Feiler 2006, p. 103].

2.2 FLOWS THROUGH COMPONENT IMPLEMENTATIONS

A flow implementation declaration in a component implementation specifies how a flow specification is realized as a sequence of flows through subcomponents along connections from the flow specification **`in`** port to the flow specification **`out`** port. The system implementation for system *S1*, shown on the right of Figure 2, contains process subcomponents *P1* and *P2*. Each process subcomponent has two ports and a flow path specification as part of its process type declaration. The implementation of flow path *F1* is shown in both graphical and textual form on the right side of Figure 2. *F1* starts with port pt1 and follows a sequence of connections and subcomponent flow specifications through connection *C1*, subcomponent flow specification *P2.F5*, connection *C3*, subcomponent flow specification *P1.F7*, and connection *C5*. This flow implementation ends with port *pt2*.

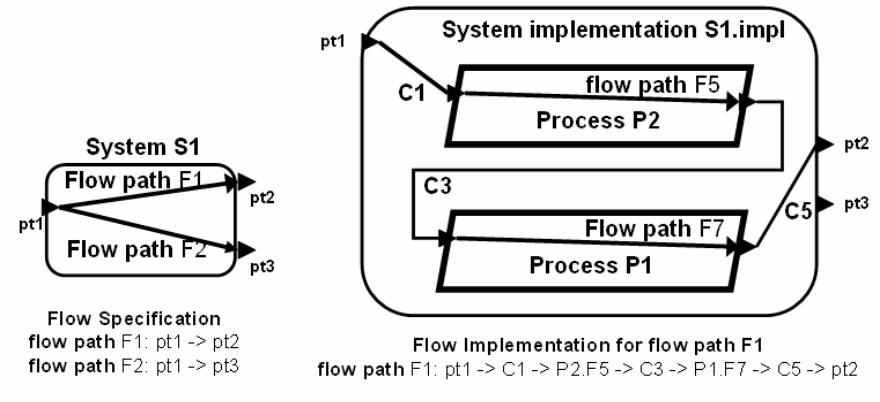

Figure 2: Flow Specification and Flow Implementation

Flow implementations can be declared for specific modes and mode transitions. Furthermore, flow implementations can have mode-specific property values, which accommodate the modeling of flows in modal systems. Once component implementations are known at multiple levels, actual flow properties such as latency at higher levels of the hierarchy can be calculated from the flow properties of the lower levels.

2.3 END-TO-END FLOWS

An end-to-end flow is a logical flow through a sequence of system components (i.e., threads, devices, and processors). An end-to-end flow is specified by an end-to-end flow declaration. End-to-end flow declarations are declared in component implementations, typically in the flow implementation in the system hierarchy that is the root of all threads, processors, and devices involved in an end-to-end flow. The subcomponent identified by the first subcomponent flow specification referenced in the end-to-end flow declaration contains the system component that is the starting point of the end-to-end flow. Subsequent named subcomponent flow specifications contain additional system components.

Figure 3 illustrates how an end-to-end flow can be graphically visualized. The text box shows all end-to-end flows of a system. The selected end-to-end flow specification is shown in black, while the other one is shown in gray. The subcomponent flows and connections that make up the selected end-to-end flow are shown in black, while subcomponent flows and connections that are not part of the selected flow are shown in gray. An editor can use this visualization to display and support the definition of end-to-end flows. The user defines the elements of an end-to-end flow by selecting and deselecting subcomponent flows and connections. Flow implementations can be visualized in a similar manner.

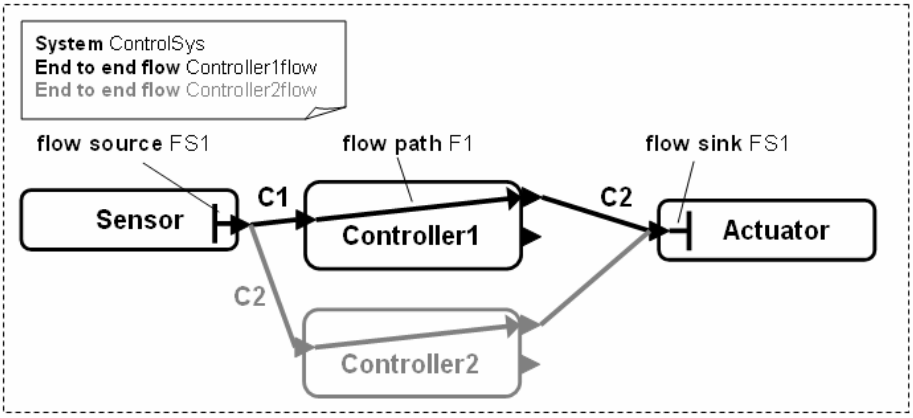

Figure 3: End-to-End Flow Declaration and Selection

Notice that an end-to-end flow is expressed in terms of the flow specifications of its subcomponents. As a result, we can analyze flows in terms of subcomponent flow specifications without requiring the implementations of those components to be specified. We can validate the property values of a flow specification using the property values derived from the flow implementation that is based on the flow specification property values of its subcomponents. This capability supports a specification-based, low-fidelity analysis of architecture models early in the life cycle, before system details are available.

2.4 END-TO-END FLOW INSTANCES

Flow declarations are associated with individual components. End-to-end flow declarations are specified in terms of the immediate subcomponents. For a system instance, these flow declarations are recursively expanded in the same way that subcomponent declarations result in a hierarchy of component instances in an AADL instance model or a collection of connection declarations results in a semantic connection.

Figure 4 shows how a flow sink specification expands into a three-level system hierarchy. The flow sink specification *FS1* for system *S2* is expanded into the connection *C1* and flow sink specification *FS2* of process *P2*, which in turn is expanded into the connection *CC1* and the flow sink specification *FS1* of thread *T5*. In short, the ultimate flow sink specification of system *S2* is the flow sink of thread *T5*.

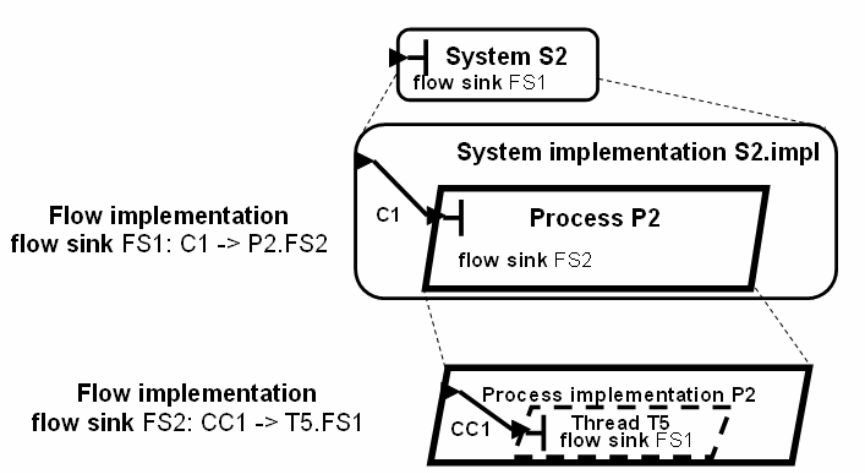

Figure 4: Flow Declarations and the System Hierarchy

Figure 5 illustrates the expansion of an end-to-end flow declaration into the end-to-end instance flow in a system instance model. Notice that the end-to-end flow declaration is declared with the component implementation that is the common root of all system components involved with the end-to-end flow. In our example, the common root is the component implementation that contains systems *S0*, *S1*, and *S2* as subcomponents. The ultimate flow source of the example end-to-end flow is the flow source in thread *T0*. The ultimate flow sink is the flow sink in thread *T5*. The end-to-end instance flow follows the semantic connection from thread *T0* to thread *T1*, the semantic connection from *T1* to *T2*, and the semantic connection from *T2* to *T3*. Note that the flow path *F1* of system *S1* represents the flow through both threads *T1* and *T2*. We have used dashed lines to mark the end-to-end instance flow in Figure 5.

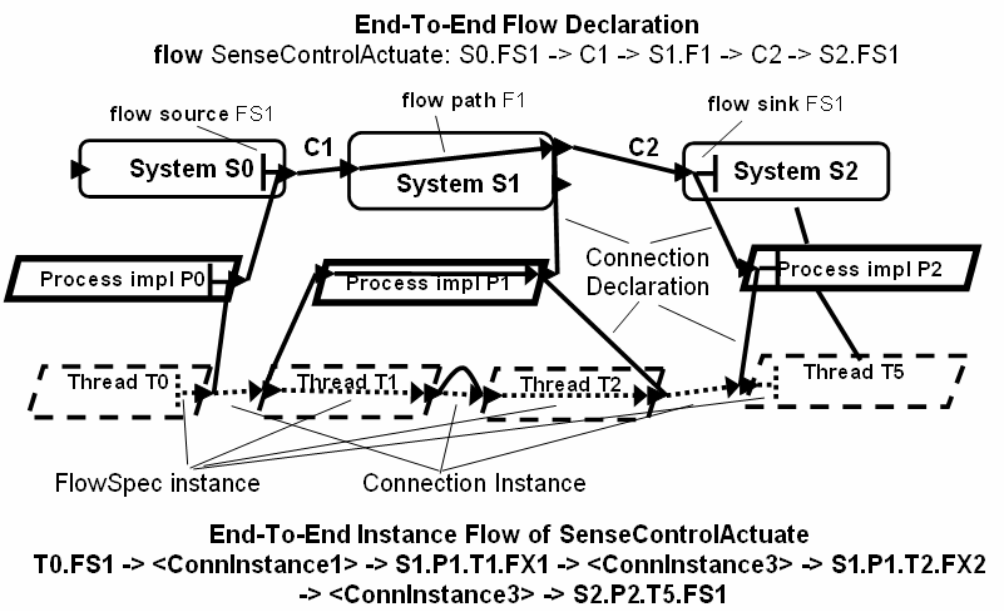

Figure 5: End-To-End Flow in a System Instance

2.5 TEXTUAL FLOW DECLARATION EXAMPLES

The example in Table 1 illustrates various aspects of defining flow specifications. The **process** `foo` has several flow specifications. `Flow1` and `Flow2` are two different flow paths through the **process** from the same incoming port to two different outgoing ports. `Flow3` represents a flow where the **process** `foo` acts as an information sink (i.e., it consumes the **in event port** `initcmd`). Similarly, the **process** `foo` acts as an information source for the **out event port** `Status`.

The **process implementation** `foo.basic` consists of several threads that are assumed to have flow specifications as indicated in the commented text,[1] and specifies a flow implementation for several flow specifications. This flow implementation indicates how the flow is realized as flow through the component's subcomponents. It also specifies two end-to-end flows that are local to the **process** `foo:` (1) `ETE1` starts with a flow source of a subcomponents and ends with a flow sink of a different subcomponent, and (2) `ETE2` specifies a flow whose starting and ending elements are flow paths (i.e., we are specifying a subflow of interest although the information flows in and out of the specified end-to-end flow).

[1] Comment lines in an AADL specification are prefaced by two dashes (--).

Table 1: Aspects of Defining Flow Specifications

```
process foo
features
   Initcmd: in event port;
   Signal: in data port gps::signal_data;
   Result1: out data port gps::position.radial;
   Result2: out data port gps::position.cartesian;
   Status: out event port;
flows
   -- two flows split from the same input
   Flow1: flow path signal -> result1;
   Flow2: flow path signal -> result2;
   -- An input is consumed by process foo through its initcmd port
   Flow3: flow sink initcmd;
   -- An output is generated (produced) by process foo and made
available
   -- through its port Status;
   Flow4: flow source Status;
end foo;
process implementation foo.basic
subcomponents
   A: thread bar.basic;
   -- bar has a flow path fs1 from p1 to p2
   -- bar has a flow source fs2 to p3
   C: thread baz.basic;
   B: thread baz.basic;
   -- baz has a flow path fs1
   -- baz has a flow sink fsink
connections
   conn1: data port signal -> A.p1;
   conn2: data port B.p2 -> result1;
   conn3: data port C.p2 -> result2;
   conn4: data port A.p2 -> B.p1;
   conn5: data port A.p2 -> C.p1;
   conn6: event port A.p3 -> Status;
   connToThread: event port initcmd -> C.reset;
flows
   Flow1: flow path
            signal -> conn1 -> A.fs1 -> conn4 ->
            C.fs1 -> conn2 -> result1;
   Flow2: flow path
            signal -> conn1 -> A.fs1 -> conn5 ->
            C.fs1 -> conn3 -> result2;
   Flow3: flow sink initcmd -> connToThread -> C.fsink;
   -- a flow source may start in a subcomponent,
   -- i.e., the first named element is a flow source
   Flow4: flow source A.fs2 -> connect6 -> status;
   -- an end-to-end flow from a source to a sink
   ETE1: end to end flow
            A.fs2 -> conn5 -> C.fsink;
   -- an end-to-end flow where the end points are not sources or
sinks
   ETE2: end to end flow
            A.fs1 -> conn5 -> C.fs1;
end foo.basic;
```

3 Latency Analysis Framework

Flow latency is the amount of time it takes for information to flow from the starting point of an end-to-end flow to its destination via connections and possibly intermediate components. The starting point, intermediate components, and destination can be threads and devices; we refer to them as tasks.

Flow latency is affected by these four factors:

1. Processing time by tasks in the end-to-end flow

   Tasks are AADL threads and devices.

2. Processing delay due to queuing or sampling

   Tasks may communicate via queued ports (AADL **`event`** or **`event data`** ports) or unqueued sampling ports (AADL **`data`** ports). The processing delay due to queuing is affected by the number of elements in the queue; the processing delay due to sampling is affected by the rate at which the information is being sampled.

3. Transfer time of information between tasks along connections

   Transfer between tasks may occur on the same processor (AADL threads bound to the same processor), between tasks on different processors, or between a task and a device. The transfer time is affected by the amount of data being transferred and the buses to which a connection is bound.

4. Transfer delay due to queuing or waiting for time slots in the transfer protocol

   Transfer delay due to queuing is affected by the number of elements in the transfer queue, while transfer delay due to time slotting is affected by the rate at which slots for transfer are made available by the transfer protocol.

The end-to-end latency of a flow is determined by the processing time of each task, processing delay by all but the first task, and transfer time and delay for each of the connections between the tasks.

We can distinguish between best-case latency and worst-case latency. Best-case latency is determined by the minimum execution time of each task involved in a flow. This establishes a lower bound under the assumption that each task can execute immediately (i.e., each task has dedicated or highest priority access to a process and is not preempted by other tasks). Worst-case latency is determined by the deadline of each task involved in the flow under the assumption that the tasks are schedulable. This establishes a lower bound. For a given periodic workload this bound can be reduced by effectively reducing task deadlines while maintaining schedulability.

Latency jitter is determined by the completion time of each task.. Completion time is affected by variation in actual execution time between the minimum and maximum execution time and is bounded by the deadline under the assumption that the task set is schedulable. For any fixed workload, the maximum completion time may be lower than the specified deadline; under varying workload, the deadline represents an upper bound under the assumption that all deadlines are met.

The age of data differs from latency, when data is re-sampled—as is the case when up-sampling occurs or there are missing data elements in the data stream (e.g., due to a missing sensor reading).

In the following sections, we look at each of these factors in more detail.

3.1 PROCESSING TIME

Tasks have several timing related properties that reflect the processing time, including the following:

- Minimum and maximum execution time

  Execution time, the amount of time the task is executing on a processor, is determined by the number of instructions executed. Therefore, execution time is dependent on processor speed and is affected by cache and pipelining techniques used by the processor. The minimum execution time can be used to determine a lower bound on latency, on the assumption that the active component is not preempted. The maximum execution time can be used to determine a lower bound on throughput, on the assumption that the active component is not preempted. `Compute_Execution_Time` plus `Recover_Execution_Time` properties in AADL specify a time range to represent these values. Processor-specific property values can be used to specify processor-type specific execution times. Alternatively, the execution time can be specified with respect to a reference processor and a scaling factor for a specific processor type is used to determine the execution time.

- Completion time and deadline

  Completion time is the amount of time that expires between the dispatch of a task and its completion. This time takes into account delays due to preemption or resource locking. The minimum completion time is the minimum execution time, under the assumption that the task execution is not delayed. Consequently, the minimum completion time is affected by the processor speed. The maximum completion time is the task deadline; it is not sensitive to the processor speed. The task deadline can be used to determine the maximum latency due to processing, on the assumption that the task set is schedulable. The `Deadline` property in AADL specifies the deadline for periodic, aperiodic, sporadic, and background threads.

For worst-case latency analysis, the AADL properties are `Deadline` and `Period` are used by the flow latency analysis tool. `Compute_Execution_Time` and `Recover_Execution_Time` are not utilized in the worst-case flow latency analysis. The lower bound of the `Compute_Execution_Time` represents the minimum execution time and represents a lowest bound for best-case end-to-end latency calculations.

3.2 PROCESSING DELAY

Several factors can cause processing delay. We examine these factors separately for sampled processing and data driven (queued) processing.

3.2.1 Sampled Processing

Sampled processing occurs when a task is dispatched independently of the arrival of the input that is processed by the task.

A periodic task may sample the input on its incoming data ports at the rate of its period. Periodic tasks may also sample **event** ports and **event data** ports at the rate of their period. The sampling rate is determined by the rate at which a task is dispatched. For periodic threads (`Dispatch_Protocol` property value `Periodic`) or devices with periodic drivers (`Device_Dispatch_Protocol` property value `Periodic`), the resulting sampling delay is the value of its `Period` property.

At each sampling point, the task may process the complete port queue, if the `Dequeue_Protocol` property value is `AllItems` (the default value), or it may process one item if the value is `OneItem`. A sampled processing **thread** may represent, for instance, a system health monitor that samples an alarm queue (**event port**) periodically. Note that if the arrival rate on an **event data port** is higher than the processing rate and only one item at a time is processed, the queue will routinely overflow and data elements will be lost.

An aperiodic or sporadic thread may sample the input on its incoming data ports if its dispatch is triggered by event or event data arrival that is not part of the flow being analyzed. The maximum interarrival rate determines the worst-case latency.

When an aperiodic or sporadic thread is dispatched by the arrival of an event or event data from its predecessor in the flow, we have queued processing (see Section 3.2.4). When a data port processes by an aperiodic thread at the completion of its predecessor, it does not introduce processing delay. Instead, we have a processing chain (see Section 3.2.3).

3.2.2 Synchronous and Asynchronous Sampling

We can distinguish between scenarios where sampling dispatches of the predecessor and a given task are performed with respect to a common clock (synchronous sampling) and those where the task dispatch is performed independently (asynchronous sampling).

In asynchronous dispatch, the dispatch rate of the sampling task determines the processing delay. A sampling task dispatch may have just missed the arrival of the output, since the output is made available independently. This circumstance results in a maximum delay of the period between dispatches of the sampling task. Figure 6 shows a periodically sampling task *T2* that samples the output of an independently executing task *T1*. The maximum processing delay due to sampling is the period of *T2*, which is added to the processing time of *T1* to determine the latency.

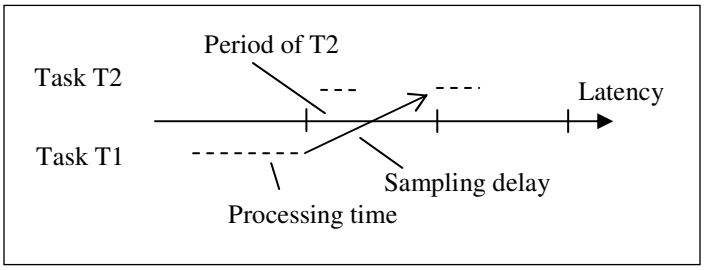

Figure 6: Asynchronous Sampling

This assumption is also the default interpretation in the AADL standard document. The AADL algorithm can easily be changed to handle globally asynchronous systems in which each processor operates with a separate clock or partially synchronous systems where some hardware components share clocks, once we have agreed on how to represent this fact through newly introduced properties.

Synchronous sampling can occur for periodic threads on the same processor and on processors and devices whose execution is coordinated by a common clock. Synchronous sampling offers a more precise figure for worst-case latency by recognizing that the execution of several tasks occurs according to a common timeline. In that situation, the originator and the sampling task are dispatched periodically and their periods are harmonic (i.e., their periods are the same or one is a multiple of the other).

Figure 7 illustrates synchronous sampling for two tasks with the same period. The time of arrival of data at the sampling task *T2* is the processing time of the originator *T1* and any transfer time and delay. The processing delay of the sampling task is the difference between the arrival of data from the synchronous predecessor and the next dispatch time of the sampling task. The latency, which is the sum of the processing time of the predecessor and the processing delay of the sampling task, is the period of *T2*.

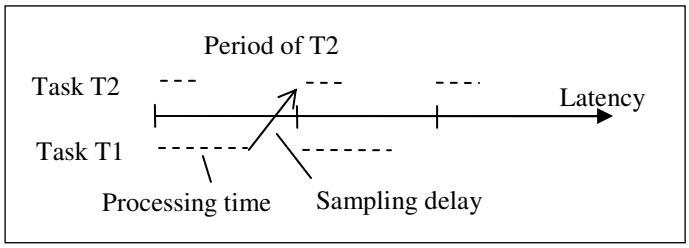

Figure 7: Synchronous Sampling

3.2.3 Sampling of Processing Chains

In general, the latency that results from synchronous sampling is the cumulative time of processing time plus any transfer time and delay rounded up to the next multiple of the sampling period of the sampling task. A periodic task may synchronously sample a processing chain that starts with a periodic task and has intermediate tasks whose dispatch is triggered by the completion of its predecessor—such as aperiodic threads or periodic threads with immediate data port connections. In this scenario, the cumulative time to be rounded up is the sum of processing times, any queued processing delay, and any transfer time and delay. Figure 8 illustrates sampling of such a processing chain *T11* and *T12* resulting in a latency of two periods of *T2*.

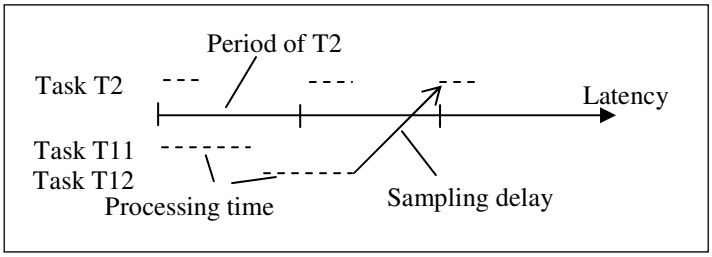

Figure 8: Synchronously Sampled Processing Chain

AADL supports data port connections with transfer timing characteristics that guarantee deterministic transfer of data streams. Data port connections can be declared to be immediate (mid-frame communication) or delayed (phase-delayed communication). We illustrate this capability in Figure 9. Delayed connections guarantee that the output of the originator is always sampled at the next dispatch of the active component, so that the processing delay is always the period of the receiver. Immediate connections delay the execution of the periodic recipient task, which effectively treats the periodic recipient as an aperiodic thread whose dispatch is triggered by the completion of the originator (i.e., a sampling processing chain).

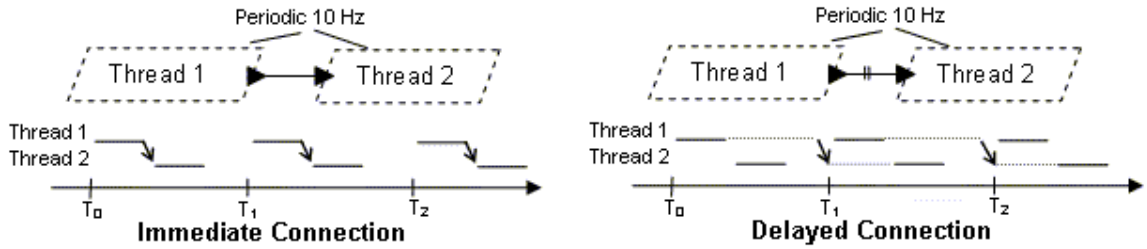

Figure 9: Timing of Immediate and Delayed Connections

3.2.4 Data-Driven Processing

Data-driven processing occurs when transferred information drives the dispatch of a task, and the dispatch request is queued if the task has an active dispatch. These are aperiodic threads or devices whose dispatch is driven by input on an event or event data port. If an end-to-end flow includes events or event data through such ports, the queuing delay contributes to the end-to-end latency. Events or event data can be sent by a thread at any time during its execution. It is assumed that in the worst case, the event or event data was sent at completion time of the predecessor. In this case, under the assumption that the system is schedulable, the predecessor's deadline is the predecessor's worst-case completion time.

The maximum processing delay is determined by the number of elements in the queue and the active dispatch. In other words, the worst-case processing delay is the queue size times the task deadline.

3.3 TRANSFER TIME AND DELAY

Transfer of information between tasks is affected by the size of the data, overhead of the transfer protocol, and the speed of the execution platform component(s) that carry out the transfer. The size of data to be transferred is specified through the `Source_Data_Size` property, which can be associated with the **`data`** or **`event data`** port or with the **`data`** `classifier` of the **`data`** or **`event data`** port.

Transfers may occur within one processor or across processors through networks/buses. Thus, the transfer time is affected by the binding of the application components and connections to the execution platform. Transfer delay is due to queuing within the transfer protocol implementation and multiplexing of the physical transfer medium using he protocol.

Several properties have been predeclared in the AADL standard for buses and processors to determine the transfer time within memory or on a bus. The predeclared properties are `Assign_Time`, `Number_of_Bytes`, `Assign_Byte_Time`, and `Assign_Fixed_Time`. Using those properties, the equation for computing transfer time within memory or on a bus is as follows: (`Assign_Time = (Number_of_Bytes * Assign_Byte_Time) + Assign_Fixed_Time`).

The AADL standard also has predeclared a `Transmission_Time` and a `Propagation_Delay` time. The latter intends to capture any queuing delay or sampling delay due time fixed time slots.

There may be situations where a hardware platform has not been identified or binding of tasks to processors has not been established. In these cases, the latency property associated with a connection is interpreted to take into account communication latency (see Section 4.6).

3.4 USE OF LATENCY PROPERTY

The AADL standard has predeclared several latency related properties:

- The `Latency` property can be specified for end-to-end flows, flow specifications, and connections. It represents the “maximum amount of elapsed time allowed between the time the data or events enter a flow or connection and the time it exits” [SAE AS5506 2004, p. 209].
- The `Expected_Latency` property specifies “the expected latency for a flow specification” [SAE AS5506 2004, p.207].
- The `Actual_Latency` property specifies “the actual latency as determined by the implementation of the end-to-end flow” [SAE AS5506 2004, p.189].

The flow latency analysis framework utilizes the `Latency` property and `Expected_Latency` property. When the analysis algorithm needs to retrieve a latency value, it first attempts to find the `Latency` value; if that value is not present, the algorithm attempts to retrieve the

Expected_Latency value. The values can be used interchangeably, with the Latency value overriding the Expected_Latency value when present.

In an end-to-end flow, the Latency (or Expected_Latency) property represents the latency value that the calculated end-to-end latency is compared against.

In a flow specification, the Latency (or Expected_Latency) property represents the flow latency that is expected to be contributed by the flow through the component. This value is used in end-to-end flow latency analysis as the latency value for each component of the instance model involved in the flow, unless the component is a thread or device and has its dispatch protocol, period, and deadline specified. If that is so, the latency determined by those property values is compared against the Latency value of the thread flow specification, and the smaller of the two is used in the end-to-end flow calculations. In other words, the Latency property value acts as a surrogate for the latency contribution by a component for which more detailed information for determining the latency contribution is not available.

When flow latency analysis is applied to a declarative AADL model, the latency is computed for each flow implementation declaration and compared with the Latency (or Expected_Latency) property value of the flow specification.

A Latency (or Expected_Latency) property value can also be associated with a connection. This value is used in end-to-end flow latency analysis by default, unless the connection is bound to a bus. In that case, the computed latency value is determined by the transfer time and transfer delay based on the binding of the connection to execution platform components (bus, processor, and device)—as discussed in Section 3.3. This computed latency can be included in the flow latency analysis by refining by redefining the getConnectionLatency method of the FlowLatencyAnalysisSwitch class to compute the latency for connection instances instead of retrieving the Latency or Expected_Latency value.

4 Latency Analysis Illustrated

In this section, we describe the calculation of end-to-end flow latency in the context of an example system to illustrate how sampling latency is determined for event and event data streams and periodic and aperiodic processing chains of threads that operate on signal streams (i.e., communicate state data through data ports). In a report on concurrency, Hansson et al. provide a more formal treatment of bounds imposed by an application task and communication architecture on latency and other preference characteristics [Hansson et al. 2007].

For periodic and aperiodic processing chains, we can treat sequences of sampling periodic threads that are dispatched with respect to a common clock as special cases for which we can determine a less conservative latency value. We can distinguish between immediate and delayed connections between periodic threads and devices. Furthermore, we can handle sampling periodic threads with different periods (i.e., threads that down-sample or up-sample) as special cases by distinguishing between harmonic threads and non-harmonic threads.

For down-sampling by harmonic threads, the lower rate receiving thread does not sample and process every data element; thus, the latency of the skipped element does not have to be considered. In up-sampling, the same data element is sampled twice (i.e., the same data value is delivered more than once). For a new value, the end-to-end latency is determined by the first sample of the new value. This data value ages as it is repeatedly sampled. If aging due to up-sampling is to be taken into account in the latency calculation, the latency is determined by the longest period in the processing chain reduced by the amount of time the last thread in the chain finishes before the end of its period (i.e., the difference between that task's period and deadline).

In the following sections, we describe the instance model on which latency analysis is performed, and introduce an example system model used in the illustration of the latency analysis. Then we discuss modeling of the flow through event data ports with aperiodic threads (data-driven processing) and periodic threads (sampled processing). We discuss flows that are represented by data ports and periodic as well as aperiodic threads. We close this section with a discussion of latency contributions by partitioned system architectures and the ability to perform multi-fidelity latency analysis.

4.1 THREAD-LEVEL INSTANCE MODELS

Thread-level instance models are fully specified instance models, whose leaf components of the component instance hierarchy are **`thread`**, **`device`**, **`processor`**, **`bus`**, and **`memory`** component instances. Communication between these components occurs through **`port`** connection instances. Port connection instances connect ports of leaf components in the instance hierarchy (e.g., from thread to thread, from device to thread, or thread to device). They represent a semantic connection, as defined in the AADL standard [SAE AS5506 2004]. A port connection instance reflects a sequence of zero or more port mappings that (1) starts with a thread or device port, (2) makes a connection from one subcomponent port to another subcomponent port, and (3) ends with a sequence of zero or more port mappings from a component port to a port of one of its subcom-

ponents. The port mappings and the subcomponent port connection are expressed by connection declarations in the component implementation that contains the subcomponent(s).

End-to-end flow instances consist of a sequence of `FlowSpec` instances of the leaf component instances involved in the flow and port connection instances that represent the flow between these leaf components. In an end-to-end flow instance, the typical sequence is as follows:

1. flow source instance
2. sequence of port connection instances and flow path instances of components in the end-to-end flow
3. port connection instance and a flow sink instance

Note that the starting and ending flow instances are not required to be flow sources and flow sinks; they can also be flow paths. This flexibility allows users to define end-to-end flows that are subsets of a complete end-to-end flow (e.g., define an end-to-end flow through the embedded software with inclusion of the flow through the sensor or actuator despite the flow of the first software component being a flow path that routes the input from a sensor to a component out port).

4.2 THE EXAMPLE MODEL

The system model in Figure 10 illustrates different aspects of the end-to-end flow analysis capability. In that figure, we present a system that consists of a sensor device *Ds*, two processes *P1* and *P2*, and an actuator device *Da*. Process *P1* consists of a single thread *T1*, while process *P2* consists of two threads *T2* and *T3*. We analyze end-to-end flows that start with *Ds*, flow through *T1*, *T2*, and *T3*, and end in *Da*. These flows may be represented by sampled **`data`** ports, or by queued **`event data`** ports. The devices and the threads may be dispatched periodically or aperiodically.

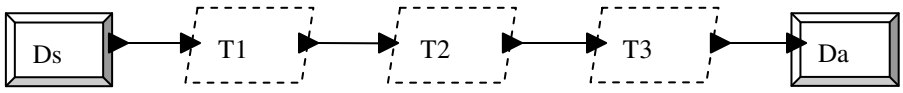

Figure 10: Flow from a Sensor through Three Threads to an Actuator

The three threads may execute on the same processor or on different processors. These processors may be connected by a bus, or they may operate with shared memory (e.g., dual-core processors). The distribution of the threads across processors may require them to be placed in separate processes. The resulting instance model is the same: port connection instances between thread instances.

The worst-case flow latency calculations described in this section represent a lower bound. In other words, distribution onto multiple processors may increase the latency due to communication. If assumptions are made about a fixed periodic workload, the lower bound may be reduced by using the maximum completion time as the effective deadline, as discussed earlier.

4.3 FLOW PROCESSING THROUGH EVENT DATA PORTS

AADL offers **`event data`** ports to support queued data or message processing. The arrival of data on such a port can trigger the execution of an **`aperiodic`** thread, which is, in effect, data-driven processing. AADL also allows event data ports to be sampled by **`periodic`** threads. In this case the thread may process one item in the queue or all items in the queue. In the remainder of this section we examine the impact of the data-driven and sampled flow representation on end-to-end latency.

The sensor device may observe an event in the physical environment and report the event through its port. This event is assumed to occur independently of any processing clock. An example of such a sensor is a one that measures the rotation of a wheel. This type of sensor activity is modeled in AADL by an aperiodic device. The sensor device may also periodically sample the physical environment, such as measuring the temperature; this type is modeled in AADL by a periodic thread.

Similarly, the actuator device may react to the data arriving at its port (i.e., behave as an aperiodic thread). An example of this type of actuator is one that adjusts the angle of a flap. Alternatively, a device may operate periodically by sampling its input port. An example of such a device is a display that refreshes at a given rate.

In the next sections, we assume that the device is an aperiodic device. The effect of periodic devices on the latency calculation is addressed in Section 4.4.4. The Appendix includes a complete AADL model example with variations of system configurations that are concrete instances of the signal flow processing variations discussed in this section.

4.3.1 Data-Driven Processing through Queued Ports

Data-driven processing is modeled in AADL by event data ports and aperiodic threads whose dispatch is triggered by the arrival of data. In the example shown in Figure 11, the devices and threads operate aperiodically. The sending of event data is triggered within the device *Ds*; the arrival of event data triggers the execution of *T1*, which in turns triggers *T2*, followed by *T3*. Completion of *T3* triggers the execution of *Da*.

We assume that *Ds* and *Da* have specified a Latency (*L*) property value for their flow specifications, while the threads have specified a Deadline (*D*). The worst-case end-to-end latency is the sum of *Ds_L* + *T1_D* + *T2_D* + *T3_D* + *Da_L*. Notice that the end-to-end flow is effectively a processing chain; its end-to-end latency is the cumulative worst-case completion time, which is the sum of the deadlines.

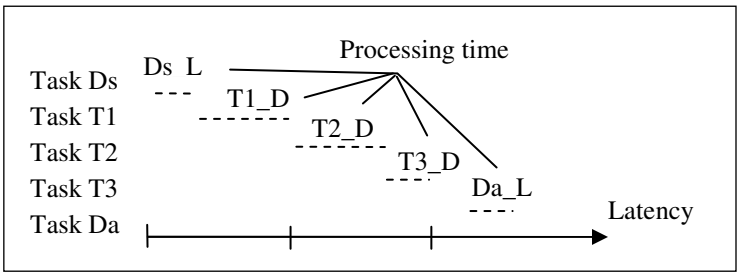

Figure 11: Data-Driven Flow Processing Chain

Table 2: Determination of Values for Data-Driven Processing

Property	Computation of Value	Detail
Worst-case flow latency	The sum of the deadlines	If the event data ports have a queue greater than zero, the latency is increased by queuing delay, which in the worst case is QueueSize * D of the thread with the incoming port.
Best-case flow latency	The sum of the minimum execution time of each processing step, with the assumption that the queues are always empty	This lower bound that increases as • faults occur and recovery execution time is added • actual execution time ranges between the minimum and maximum values • completion time increases due to preemption by other threads
Maximum latency jitter	The sum of • the difference between the minimum execution time and the deadline of all threads • the sum of queuing latencies	This variation can be larger than the period of an individual processing step (frame).
Age of data	Same as its latency	

Observations

- If a thread has a specified flow specification latency, this latency is expected not to exceed the deadline of the thread. If a specified flow latency smaller than the deadline can be guaranteed, effectively making it the deadline of the thread and assuring schedulability, that flow latency can be used in the calculation.
- All sensor readings are processed by all processing steps unless a port queue overflows. Missed sensor readings result in a missed output in the end-to-end flow.

4.3.2 Sampled Data Stream Processing through Periodic Threads

Event data ports that are sampled by periodic threads can be used to represent applications such as a health monitor that periodically monitors event, alarm, or message streams without creating overload conditions (due to high burst alarm) or message rates (by not reacting to every arriving event or event data individually). In a sampled data stream, the thread is assumed to examine all items in the port queue on dispatch.

In the example shown in Figure 12, the devices operate aperiodically, while the threads periodically sample the flow through event data ports. The threads may sample with the same rate (period) or different rates.

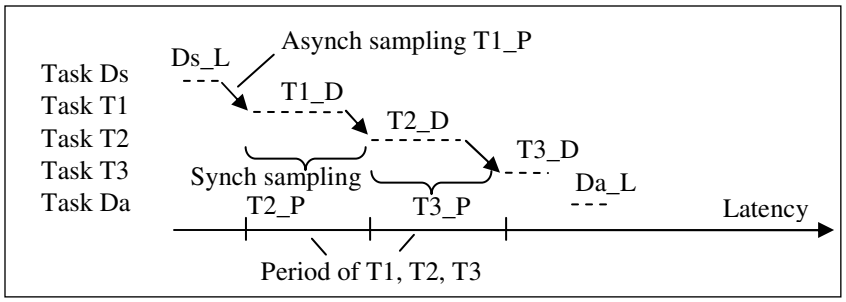

Figure 12: Synchronous Sampling Task Sequence

Table 3: Determination of Values for Sampled Data Stream Processing

Property	Computation of Value	Detail
Syncrhonous sampling (with all sampling at same rate)		
Worst-case flow latency	The sum of *Ds_L* + *T1_P* + *T2_P* + *T3_P* + *T3_D* + *Da*_L.	For *T2* and *T3*, the sampling delay *Ti_P* is assumed to be larger or equal to *Ti-1_D*. This is the case if all the periods are the same and the deadline is less or equal to the period.
Best-case flow latency	The sum of *Ds_L* + *T1_P* + *T2_P* + *T3_P* + *T3_D* + *Da*_L.with the deadline replaced by the minimum execution time	
Maximum latency jitter	The difference between the minimum execution time and the deadline of *T3*, which is less than the period of *T3*.	It is within a frame.
Age of data	The same as the latency unless an element is missing in the data stream	For each set of consecutively missing elements the age increases by the time interval since the last real value of the component that drops a data stream element. This may be the sensor or any of the processing steps. For example, if the sensor operates at a rate of 10 ms, every missed reading adds 10ms. If the computation of T2 operates at 20 ms and is unable to produce output in time, 20 ms are added to the age of the data passed to the actuator. There is no increase in age due to up-sampling in this scenario, as the threads are assumed to have the same period.
Output	Produced with every period	It may be based on aged data.
Asynchronous sampling (the dispatch of different threads is triggered by different clocks)		
Worst-case flow latency	The sum of *Ds_L* + *T1_P* + *T1_D* + *T2_P* + *T2_D* + *T3_P* + *T3_D* + *Da_L*.	Clocks may be offset from each other and have clock drift. The maximum offset is equal to the period; thus, we add the sampling period to the processing time of the predecessor.
Best-case flow latency	The sum of *Ds_L* + *T1_P* + *T1_D* + *T2_P* + *T2_D* + *T3_P* + *T3_D* + *Da_L*. with the deadline replaced by the minimum execution time	
Maximum latency jitter	The sum of the differences between the minimum execution time and the deadline of each thread .	It may exceed one or more frames.
Age of date	The same as the latency unless an element is missing in the data stream	
Output	Produced with every period of *T3*	It may be based on aged data.

4.3.3 Mixed Event Data Flow Processing

In mixed event data flow processing, the devices operate aperiodically, while processing is a combination of sampled and data-driven processing. In other words, the threads are a combination of aperiodic and periodic threads. In our example, we assume that the first thread is aperiodic; the second, periodic; and the third, aperiodic.

The cumulative processing time of the sensor device *Ds* and the thread *T1* are sampled by the thread *T2*. This sampling is asynchronous because it is the first sampling in the flow. The processing times of *T2*, *T3*, and the actuator device *Da* are cumulative and add to the total latency.

The worst-case end-to-end latency is the sum of *Ds_L* + *T1_D* + *T2_P* + *T2_D* + *T3_D* + *Da_L*.

The best-case latency for this scenario is the above formula with the deadline replaced by the minimum execution time.

The maximum latency jitter is the sum of differences between the minimum execution time and the deadline of *T1*, *T2*, and *T3*.

The age of data is determined the same way as in section 4.3.2.

Output is produced with every period, but may be based on aged data.

A variant of this scenario is that threads *T1* and *T3* are periodic, while thread *T2* is aperiodic. In that variation, *T1* performs asynchronous sampling of the *Ds* processing time. The cumulative processing time of *T1* and *T2* is sampled by *T3*. This sampling is synchronous with respect to the period of T1.

The end-to-end latency in this variant is the sum of *Ds_L* + *T1_P* + (*T1_D* + *T2_D*)>*T3_P* + *T3_D* + *Da_L*. The notation (*X*)>*Y* indicates that the value *X* is rounded up to the next multiple of *Y*. In this scenario the periods of *T1* and *T3* may differ.

The best-case latency for this scenario is the above formula with the deadline replaced by the minimum execution time.

The maximum latency jitter is the difference between the minimum execution time and the deadline of *T3* plus zero or more multiples of the *T3* period, because the sum of minimum execution time of *T1* and *T2* rounds up to *X* multiple of *T3_P*, while the sum of their deadlines rounds up to *Y* multiple of *T3_P* with *X* less or equal to *Y*. In practical terms this means that the signal stream is sampled non-deterministically and as a result the latency may oscillate by multiples of the period of *T3*. Non-deterministic sampling refers to the fact that the sampling thread may sometimes sample the same data value twice while at other times skip a data value.

The age of data is determined the same way as in section 4.3.2.

Output is produced with every period of *T3*, but may be based on aged data.

If sampling occurs asynchronously, then the worst-case latency is the sum of *Ds_L* + *T1_P* + *T1_D* + *T2_D* + *T3_P* + *T3_D* + *Da_L*

The best-case latency for the asynchronous case is the above formula with the deadline replaced by the minimum execution time.

The maximum latency jitter is the sum of differences between the minimum execution time and the deadline of T1, T2, and T3, which may exceed one or more frames.

The age of data for the asynchronous case is determined the same way as in section 4.3.2.

Output is produced with every period of T3, but may be based on aged data.

4.3.4 Harmonic Up and Down Sampling

In harmonic sampling, threads have different periods. Processing along the flow may perform down- and up-sampling. For example, *T1* may sample at 40Hz, *T2* at 20Hz, and *T3* at 40Hz. In this case, we have harmonic periods among sampling threads. If successive threads have harmonic periods, we can utilize the synchronous sampling reduction.

The worst-case end-to-end latency, then, is the sum of *Ds_L* + *T1_P* + *T1_D*>*T2_P* + *T2_D*>*T3_P* + *T3_D* + *Da_L* (see Figure 13). This formula indicates the rounding up of *T2's* processing time to the next multiple of *T3's* sampling delay. If the thread deadlines are equal to the periods, then *T1_D*>*T2_P* and *T2_D*>*T3_P* have the value *T2_P*, since *T2_P* is a multiple of *T1_P* and *T3_P*.

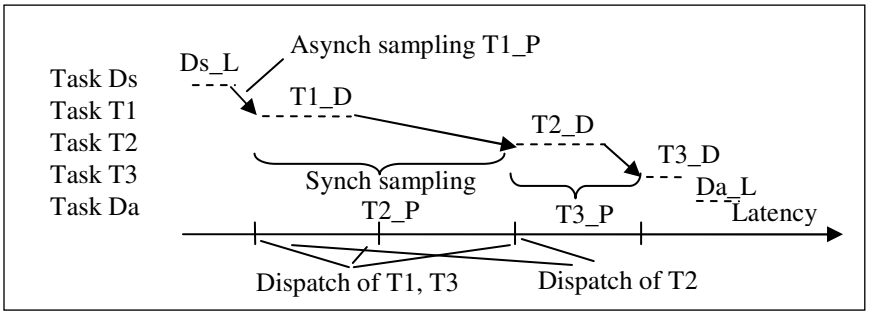

Figure 13: Harmonic Sampling

The best-case latency for this scenario is the above formula with the deadline replaced by the minimum execution time. Note that the completion time of *T2* varies between its minimum execution time and its deadline, which may span two periods of *T3*. The sampling latency contribution of *T3* may be one or two times its period. In other words, *T3* may sample non-deterministically.

The maximum latency jitter is the difference between the minimum execution time and the deadline of *T3* plus the sampling variation of one *T3* period due to up-sampling, the total being more than one frame. In case of 2X up-sampling the sampling thread may sample the same element three times and then an element only once instead of sampling every element twice.

If sampling occurs as part of the application code instead of being performed by the runtime system at the time of dispatch, *T2* may also sample non-deterministically because sampling occurs with the execution of the first instructions. This time can vary depending on the execution of other threads. In case, of 2X up-sampling this means that instead of sampling every other data element

the down-sampling thread may sample two data elements in a row and then skip two data elements – adding to the latency jitter.

The age of data is determined the same way as in Section 4.3.2. Due to up-sampling by *T3* the age can increase by the period of *T3*. In such a case it is common practice to use extrapolation to avoid this age increase.

Output is produced with every period of *T3*, but may be based on aged data.

If sampling occurs asynchronously (i.e., the dispatch of different threads is triggered by different clocks), then the worst-case end-to-end latency is the sum of *Ds_L* + *T1_P* + *T1_D* + *T2_P* + *T2_D* + *T3_P* + *T3_D* + *Da_L*. Clocks may be offset from each other and have clock drift. The maximum offset is equal to the period, thus, we add the sampling period to the processing time of the predecessor.

In the asynchronous case, the best case latency is also determined by substituting the minimum execution time for the deadline.

In this case, the maximum latency jitter is the sum of the differences between the minimum execution time and the deadline of each thread, plus any jitter due to non-deterministic sampling. This jitter variation is higher than that of the synchronous case.

The age of data is determined the same way as in Section 4.3.2. Due to up-sampling by *T3* the age can increase by the period of *T3*. In such a case it is common practice to use extrapolation to avoid this age increase.

Output is produced with every period of *T3*, but it may be based on aged data.

4.3.5 Non-harmonic Sampling

In non-harmonic sampling, the thread periods differ and their values are non-harmonic. Non-harmonic sampling occurs if the periods of two successive threads are not multiples of each other.

In the case of non-harmonic synchronous sampling, the latency by one processing step is the sum of the processing time *Ti-1_D* and the sampling (processing) delay *Ti_P* reduced by the largest common multiple (LCM) of the periods *Ti-1_P* and *Ti_P*. The reduction is because the LCM represents the time step by which the dispatch times of the non-harmonic thread dispatches differ along the timeline.

4.4 FLOW PROCESSING THROUGH DATA PORTS

AADL offers **`data`** ports to support sampled processing of data (i.e., processing of the most recent data value). Sampling occurs through periodic threads. For data port connections between **`periodic`** threads, the AADL assures deterministic sampling through immediate and delayed port connections. Immediate port connections assure mid-frame communication, while delayed connections assure frame-delayed communication. The AADL also allows threads with data ports to be triggered by events. In particular, a thread can be dispatched as result of completion of the predecessor thread. This is specified in AADL by an event connection from the predeclared `Complete` port of the predecessor to the `Dispatch` port of the thread to be dispatched. This allows threads with data ports to be used for data-driven processing.

For our discussion in the next sections, we assume that the device is an aperiodic device. The effect of periodic devices on the latency calculation is addressed in Section 4.4.4. The Appendix includes a complete AADL model example with variations of system configurations that are concrete instances of the signal flow processing variations discussed in this section.

4.4.1 Use of Immediate Connections

In this scenario, all threads execute periodically and are connected by immediate connections. This means that the execution of *T2* is delayed until *T1* completes and passes its output on. Similarly, *T3* delays its execution until *T2* completes and passes its output on. The result is mid-frame communication between *T1*, *T2*, and *T3*. Note that processing of all three threads must complete by the deadline of *T3*.

The worst-case end-to-end latency is computed as follows: *Ds_L* + *T1_P* + *T3_D* + *Da_L*. *T1* is still sampling the sensor, while *T1*, *T2*, and *T3* form a processing chain with a common dispatch time and a deadline of *T3_D*.

The best-case latency in this scenario is *Ds_L* + *T1_P* + sum of minimum execution time for *T1*, *T2*, and *T3* + *Da_L*.

The maximum latency jitter is less than a frame, the difference between the sum of minimum execution times of all three threads and the deadline of *T3*.

The age of data is determined the same way as in Section 4.3.2, that is, as an addition equivalent to between the last actual data value from the sensor and the missed element(s).

Output is produced with every period of *T3*, but it may be based on aged data.

If processing is distributed across processors with different clocks, latency increases by the communication delay. It is not affected by clock offset or draft because the immediate connection sequence effective acts like data-driven processing.

4.4.2 Use of Delayed Connections

In this scenario all threads execute periodically and are connected by delayed connections. This means that the output of *T1* is delayed until its deadline. *T2* samples the output of *T1* relative to *T1*'s deadline rather than its completion time. Similarly, *T3* samples the output of *T2* relative to *T2*'s deadline rather than its completion time. The result is frame-delayed communication between *T1*, *T2*, and *T3*.

The worst-case end-to-end latency is computed as follows: *Ds_L* + *T1_P* + *T1_D*>*T2_P* + *T2_D*>*T3_P* + *T3_D* + *Da_L*. *T1* is still sampling the sensor, while *T1*, *T2*, and *T3* form a processing chain with guaranteed frame-delayed communication.

The best-case latency in this scenario differs from the worst-case in that *T3_D* is replaced by the minimum execution time for *T3*.

The maximum latency jitter is less than a frame, the difference between the minimum execution time and the deadline of *T3*.

The age of data is determined the same way as in Section 4.3.2, that is, as an additional equivalent to between the last actual data value from the sensor and the missed element(s).

Output is produced with every period of *T3*, but may be based on aged data.

If processing is distributed across processors with different clocks, latency increases by clock offset and drift (i.e., by a maximum of *T2_P* and *T3_P*). The jitter increases by the clock drift delta for *T2* and *T3*.

4.4.3 Mixing Immediate and Delayed Connections

The flow through data ports can be a combination of immediate and delayed connections. In Figure 14, we assume the connection *T1* -> *T2* is immediate while *T2* ->> *T3* is delayed. *T1* samples the output of the device; and *T3*, the output of the processing chain *T1-T2*.

The worst-case end-to-end latency is the sum of *Ds_L* + *T1_P* + (*T2_D*)>*T3_P* + *T3_D* + *Da_L*, with the sampling delay of *T3* rounded up the deadline of *T2*. Notice that when *T1* and *T2* are dispatched at the same time, the latest completion time of this processing chain is the deadline of the last element in the processing chain (*T2* in our example).

The best-case latency in this scenario differs from the worst-case in that *T3_D* is replaced by the minimum execution time for *T3*.

The maximum latency jitter is less than a frame, the difference between the minimum execution time and the deadline of *T3*.

The age of data is determined the same way as in section 4.3.2, that is, as an addition equivalent to between the last actual data value from the sensor and the missed element(s).

Output is produced with every period of *T3*, but may be based on aged data.

If processing is distributed across processors with different clocks, latency increases by clock offset and drift between the clocks of *T1* and *T3* (i.e., by a maximum of *T3_P*). The jitter increases by the clock drift delta for *T3*.

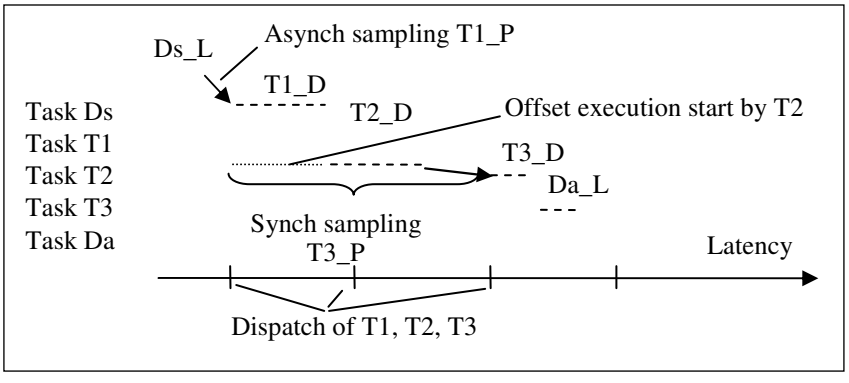

Figure 14: The Effect of Immediate Data Port Connections on Latency

If the thread *T2* is aperiodic and dispatched by the completion of its predecessor *T1* instead of an immediate connection, the cumulative time being sampled by *T3* is the sum of *T1_D* and *T2_D*.

This time, rounded up to the next period of *T3*, is the worst-case latency contributor; the best-case latency contributor is the sum of minimum execution times of *T1* and *T2*. The resulting maximum latency jitter is larger than that for periodic *T2* with an immediate connection.

4.4.4 Data-Driven Processing of Data Ports

The use of **`data`** ports for transferring data and triggering the execution of each aperiodic thread through the completion event of the predecessor thread is equivalent to the use of event data ports with queue size of zero or one.

4.5 USE OF PERIODIC DEVICES

We have assumed that devices do not operate periodically, the sensor/input device reading is triggered by an external event, and the actuator/output is processed at the time of arrival of the data.

In the AADL standard, the sensor device can be declared to operate periodically (e.g., a temperature sensor reading the temperature every second) through the `Device_Dispatch_Protocol` property, whose default value is `Aperiodic`. In the periodic case, we assume that the **`device`** and the **`processor`** executing the **`thread`** operate from a single global clock and the analysis plug-in applies the synchronous sampling reduction. We also assume that the **`device`** has a `Period` and a `Deadline` defined. The processing time of *Ds*—being either the latency or deadline—is synchronously sampled by periodic *T1* resulting in *Ds_L>T1_P* as the first contributor to the end-to-end latency. If *T1* is aperiodic or periodic with an immediate connection coming from the device, *Ds* and *T1* act as a processing chain resulting in (*Ds_L* + *T1_D*)>*T2_P* as the first latency contributor.

The actuator device may also execute periodically by sampling the output of *T3* to drive a physical device. In this case, the last contribution is *T3_D*>*Da*_P + *Da_D*, assuming that the actuator device has a period and deadline specified.

4.6 COMMUNICATION LATENCY

The previously mentioned formulas have not included communication latency, which is determined as discussed in Section 3.3. For synchronous sampling, if the sum of the processing time of the sender thread (i.e., its deadline) and the communication latency do not exceed the period of the recipient, the communication latency does not add to the end-to-end latency. This circumstance may occur when a system architect sets the deadline of a sending thread to be before the end of the period by an amount that is the maximum expected communication latency.

If the sum of the sender processing time and the communication latency exceeds the period of the recipient in synchronous sampling, the result is a sampling latency of the next multiple of the recipient period. In this case, the maximum latency jitter is affected by the communication latency.

For asynchronous sampling, the communication latency directly contributes to the end-to-end latency in the same way as sender processing time.

4.7 PARTITIONED SYSTEMS

Some system architectures introduce time and space partitioning through the concept of a partition (see ARINC 653 [ARINC653 2006]). Partitions represent virtual processors that themselves are responsible for scheduling the execution of threads, resulting in the virtualization of the timeline of threads within a partition. Partition execution order can affect latency. In order to maintain predictability and determinism of communication timing and latency and isolate the application from partition allocation to processors, interpartition communication is expected to occur in a phase-delayed fashion (i.e., the data arrives at the recipient partition at its next partition period).

We have introduced a `Partition_Latency` property that reflects the latency contribution of a partition in cross-partition communication. This latency corresponds to the period at which a partition is executed on a processor. In addition, an `Is-Partition` Boolean property is provided to indicate whether a system or process should be interpreted as a partition. Toggling the `Is_Partition` property value allows for "what-if" analysis without having to reenter the partition period value.

The flow latency analysis plug-in takes into account these two partition properties in determining the end-to-end latency in partial system models as well as system models that have been expanded to the thread level. Cross-partition communication essentially has the effect of a sampling delay on the order of `Partition_Latency`.

In a periodic recipient thread, the sampling latency is the larger of the partition period or the period of the thread. A thread executing at a slower rate than the partition execution drives the sampling. Where a thread executes at a higher rate than the partition (i.e., multiple thread executions occurring in the same partition dispatch), the partition drives the sampling.

For synchronous execution of partitions, the cumulative time includes the communication latency and is rounded up to the next multiple of the sampling latency. For asynchronous execution of partitions (i.e., execution based in independent clocks), the cumulative time including communication latency and the sampling latency are added to the total latency. In both cases, the cumulative time is reset to zero.

Given the assumption that interpartition communication is always delayed to the partition period, the latency contribution of such communication is determined independently of the binding to the execution platform. Consequently, the latency of interpartition communication can be taken into account for system models that do not include execution platform components or bindings to the execution platform.

4.8 MULTIPLE FIDELITY LATENCY ANALYSIS

Systems may be modeled at various levels of fidelity. Early in the design process, a system may be modeled in terms of one or two layers of subsystems. A system integrator may model a system of systems in terms of its systems without detailed models of each of those systems.

AADL and the OSATE toolset allow such models to be instantiated and analyzed. This compatibility allows us to support end-to-end latency analysis of partial models, where an end-to-end flow

declaration results in an end-to-end flow instance specifying a flow through the system or process components that are the leaves of the instance model.

In a partially specified instance model, the component instance hierarchy is expanded as much as possible. Expansion stops if a subcomponent does not have a classifier, has only a component type, or has a component implementation without subcomponents. Feature (port) instances are added to component instances for which the component type is defined. Port connection instances are created between the port instances of the lowest component instances with port instances. Typically those are the leaf component instances unless the leaf component instance does not have a classifier, in which case the direct parent is considered the leaf node with port instances. These port connection instances do not represent semantic connections as defined in the standard because they do not connect threads, processors, and devices. However, these instances permit partially specified instance models to be processed as low fidelity models of a system.

For example, a system may be modeled in terms of subsystems that get mapped into separate partitions in a partitioned architecture. We can perform worst-case end-to-end analysis taking into account the sampling latency due to interpartition communication.

If the subsystem flow specifications include a latency property, the expected latency due to processing within a subsystem can be taken into account. If the connections have a latency property, communication latency is taken into account as well.

When at least one subsystem has been elaborated down to the thread level, the end-to-end latency analysis can be revisited. For an elaborated subsystem, the latency calculation takes into account periodicity, sampled and data-driven processing, and other latency contributors. This analysis identifies the offending subsystem, if the end-to-end latency increases compared to the subsystem-level analysis. For example, an application system may perform its communication through an application-level high priority *periodic I/O* task that receives input from other subsystems and places it into an internal data area. Similarly, the system may take output from an internal data area and pass it on to other subsystems at the beginning of the next frame. If such an application is ported to a partitioned architecture, the end-to-end latency contribution due to interpartition communication may double, since the partition communication mechanism adds sampling communication delay and the application-level *periodic I/O* thread adds sampling delay.

5 The Flow Latency Analysis Plug-in

We have provided a plug-in to OSATE that performs worst-case flow latency analysis. Its implementation currently has some restrictions that are outlined below. This plug-in can easily be extended to perform best-case and jitter analysis as well as calculation of age of data.

The following restrictions currently (OSATE release 1.5.1) hold for the implementation of the Flow Latency Analysis plug-in:

- Sampled processing delay
  - The Flow Latency Analysis plug-in currently assumes that the `Dequeue_Protocol` value is `AllItems`.
  - The Flow Latency Analysis plug-in currently does not support independent sampling of a flow by an aperiodic or sporadic thread. The plug-in assumes that an aperiodic thread is dispatched by a completion event or event data output from its predecessor in the flow (i.e., we have queued processing).
- Synchronous vs. asynchronous sampling
  - The Flow Latency Analysis plug-in currently assumes that the execution platform is globally synchronous (i.e., periodic threads and devices are dispatched by a common clock).
- Communication latency
  - The Flow Latency Analysis plug-in currently does not take into account any execution platform properties or the size of the data being transferred. Instead, it interprets the latency property associated with a connection to take into account communication latency (see Section 4.6). The Flow Latency Analysis plug-in can easily be extended by redefining the `getConnectionLatency` method of the `FlowLatencyAnalysisSwitch` class to calculate the latency based on the other properties.
  - The Flow Latency Analysis plug-in currently does not compute the communication latency value from bus properties such as the transfer time and transfer delay based on the binding of the connection to execution platform components (bus, processor, and device)—as discussed in Section 3.3. Instead it uses the latency value associated with the connection, which represents a default value that is independent of a specific hardware binding and could represent communication within a processor. This computed latency can be included in the flow latency analysis by refining by redefining the `getConnectionLatency` method of the `FlowLatencyAnalysisSwitch` class to compute the latency for connection instances instead of retrieving the `Latency` or `Expected_Latency` value.
- Latency property use
  - The calculated end-to-end flow latency is used in the comparison and recorded as a result through the report mechanism but is currently not explicitly stored back into the AADL model as an `Actual_Latency` value.

- Aging of data
  - The flow latency analysis tool does not include aging in its latency calculation.
- Non-harmonic synchronous sampling
  - The latency analysis plug-in currently does not take into consideration this reduction. Instead, the plug-in assumes asynchronous sampling (i.e., it uses a slightly more conservative latency value).

The plug-in also supports the validation of flow specification latency values by comparing them against flow implementation latency calculations. The flow implementation latency is calculated in terms of the immediate subcomponents in the component hierarchy, not in terms of the leaf components. The flow specification latency property value of the immediate subcomponent and its partition latency property, and the connection latency are used in the calculation.

6 Summary

In this report, we introduced an end-to-end latency analysis framework that operates on AADL models. This latency analysis framework allows us to determine worst-case and best-case end-to-end latency and age of signal data streams as well as variation in latency and age. Control systems are signal processing applications that are sensitive to such latency jitter. This analysis helps identify whether deployment and porting of control system applications to different hardware platforms and runtime system architectures will result increased instability of the control algorithms.

The analysis framework identifies all contributors to latency and latency jitter. The algorithms for calculating worst-case and best-case end-to-end latency and latency variation have been illustrated in the context of a specific system model. Data-driven and sampling application architectures, different choices of communication mechanisms, and impact of partitioned architectures has been taken into account in the latency calculation.

A flow latency analysis plug-in is available as part of OSATE, an open source toolset for AADL. This plug-in currently supports worst-case latency analysis and can easily be extended to support best-case and jitter analysis.

Appendix: Example AADL Model

Sampled Processing and Data-Driven with Event Data Ports

```
-- this file contains a model that illustrates end-to-end latency due to sampled processing
-- sampled processing occurs through event data port communication.
-- The event data ports are configured to be of queue size one with the latest data in the queue.
-- the example is a signal flow from a sensor through three processing steps to an actuator.
-- The first and third processing steps operate at twice the rate of the second step.
-- The steps have a compute execution time that can vary between the specified ranges.
-- The sensor device is the originator of the signal stream.
-- The sensors operate under two scenarios:
-- 1) the sensor periodically probes the environment, i.e., executes periodically.
-- 2) the sensor reading is triggered by some physical event that occurs randomly with a maximum rate.
-- The sampling latency is affected by whether the system operates with respect to a global clock (synchronous system) or independent clock (asynchronous system).
-- The models below are set up to execute under a synchronous and an asynchronous system.

data timedata
end timedata;

-- the processing steps are defined as threads inside processes.
-- this allows them to be distributed onto different processors or execute on the same processor.
-- The threads are periodic threads that use event data port connections to sample at dispatch time.
-- This controls the amount of jitter in end-to-end latency.
--
-- In a separate model we will describe the same architecture that samples the data stream deterministically.

-- Step1 executes at a rate of 20 Hz and has a deadline or maximum latency of 45 ms.
thread step1
features
  ined: in event data port timedata { Queue_Size => 0; };
  outed: out event data port timedata;
flows
  flow1: flow path ined -> outed { latency => 45 ms;};
properties
  period => 50 ms;
  deadline => 45 ms;
```

```
  Compute_Execution_Time => 6 ms .. 10 ms;
end step1;

thread implementation step1.periodic
flows
  flow1: flow path ined -> outed;
properties
  Dispatch_Protocol => Periodic;
end step1.periodic;

thread implementation step1.aperiodic
flows
  flow1: flow path ined -> outed;
properties
  Dispatch_Protocol => Aperiodic;
end step1.aperiodic;


thread step2
features
  ined: in event data port timedata { Queue_Size => 0; };
  outed: out event data port timedata;
flows
  flow1: flow path ined -> outed { latency => 70 ms;};
properties
  period => 100 ms;
  deadline => 70 ms;
  Compute_Execution_Time => 15 ms .. 23 ms;
End step2;

thread implementation step2.periodic
flows
  flow1: flow path ined -> outed;
properties
  Dispatch_Protocol => Periodic;
end step2.periodic;

thread implementation step2.aperiodic
flows
  flow1: flow path ined -> outed;
properties
  Dispatch_Protocol => Aperiodic;
end step2.aperiodic;


thread step3
features
  ined: in event data port timedata { Queue_Size => 0; };
  outed: out event data port timedata;
flows
  flow1: flow path ined -> outed { latency => 45 ms;};
```

```
properties
  period => 50 ms;
  deadline => 45 ms;
  Compute_Execution_Time => 6 ms .. 10 ms;
End step3;

thread implementation step3.periodic
flows
  flow1: flow path ined -> outed;
properties
  Dispatch_Protocol => Periodic;
end step3.periodic;

thread implementation step3.aperiodic
flows
  flow1: flow path ined -> outed;
properties
  Dispatch_Protocol => Aperiodic;
end step3.aperiodic;

-- at the beginning of each dispatch the sensor device reads the
clock
-- and passes it as the value of its output
device sensor
features
  outed: out event data port timedata;
  devbus: requires bus access devicebus;
flows
  flow1: flow source outed { latency => 2 ms;};
properties
  period => 50 ms;
  deadline => 2 ms;
  Compute_Execution_Time => 1 ms .. 2 ms;
end sensor;

-- sensor periodically senses the physical environment
device implementation sensor.periodic
flows
  flow1: flow source outed;
properties
  Device_Dispatch_Protocol => Periodic;
end sensor.periodic;

-- sensor detects an in the physical environment
-- the occurs randomly with a maximum rate of the period
device implementation sensor.aperiodic
flows
  flow1: flow source outed;
properties
  Device_Dispatch_Protocol => Aperiodic;
end sensor.aperiodic;
```

```
-- The actuator will read the clock
-- and log the difference to the received data (sensor clock time) as
its last action
device actuator
features
  ined: in event data port timedata { Queue_Size => 0; };
  devbus: requires bus access devicebus;
flows
  flow1: flow sink ined { latency => 3 ms;};
properties
  period => 50 ms;
  deadline => 3 ms;
  Compute_Execution_Time => 1 ms .. 3 ms;
end actuator;

-- output is sampled. This reduces the latency jitter.
device implementation actuator.periodic
flows
  flow1: flow sink ined;
properties
  Device_Dispatch_Protocol => Periodic;
end actuator.periodic;

-- arrival of data causes actuator to become active.
-- This reduces end-to-end latency at the expense of increased lit-
ter.
device implementation actuator.aperiodic
flows
  flow1: flow sink ined;
properties
  Device_Dispatch_Protocol => Aperiodic;
end actuator.aperiodic;

process Pstep1
features
  ined: in event data port timedata;
  outed: out event data port timedata;
flows
  flow1: flow path ined -> outed;
end Pstep1;

process implementation Pstep1.periodic
subcomponents
  Tstep1: thread Step1.periodic;
connections
  cin: event data port ined -> Tstep1.ined;
  cout: event data port Tstep1.outed -> outed;
flows
  flow1: flow path ined -> cin -> Tstep1.flow1 -> cout -> outed;
end Pstep1.periodic;
```

```
process implementation Pstep1.aperiodic
subcomponents
  Tstep1: thread Step1.aperiodic;
connections
  cin: event data port ined -> Tstep1.ined;
  cout: event data port Tstep1.outed -> outed;
flows
  flow1: flow path ined -> cin -> Tstep1.flow1 -> cout -> outed;
end Pstep1.aperiodic;


process Pstep2
features
  ined: in event data port timedata;
  outed: out event data port timedata;
flows
  flow1: flow path ined -> outed;
end Pstep2;

process implementation Pstep2.periodic
subcomponents
  Tstep2: thread Step2.periodic;
connections
  cin: event data port ined -> Tstep2.ined;
  cout: event data port Tstep2.outed -> outed;
flows
  flow1: flow path ined -> cin -> Tstep2.flow1 -> cout -> outed;
end Pstep2.periodic;

process implementation Pstep2.aperiodic
subcomponents
  Tstep2: thread Step2.aperiodic;
connections
  cin: event data port ined -> Tstep2.ined;
  cout: event data port Tstep2.outed -> outed;
flows
  flow1: flow path ined -> cin -> Tstep2.flow1 -> cout -> outed;
end Pstep2.aperiodic;


process Pstep3
features
  ined: in event data port timedata;
  outed: out event data port timedata;
flows
  flow1: flow path ined -> outed;
end Pstep3;

process implementation Pstep3.periodic
subcomponents
  Tstep3: thread Step3.periodic;
connections
```

```
  cin: event data port ined -> Tstep3.ined;
  cout: event data port Tstep3.outed -> outed;
flows
  flow1: flow path ined -> cin -> Tstep3.flow1 -> cout -> outed;
end Pstep3.periodic;

process implementation Pstep3.aperiodic
subcomponents
  Tstep3: thread Step3.aperiodic;
connections
  cin: event data port ined -> Tstep3.ined;
  cout: event data port Tstep3.outed -> outed;
flows
  flow1: flow path ined -> cin -> Tstep3.flow1 -> cout -> outed;
end Pstep3.aperiodic;


system application
features
  db: requires bus access devicebus;
end application;

-- this application configuration has all processing steps as well as
the sensor and actuator as periodic tasks.
-- The connections are delayed connections to allow for deterministic
sampling at each step.
-- The worst-case end-to-end latency for this system on a synchronous
execution platform is the
-- sum of the periods of the three processing steps plus the actuator
period (sampling latencies)
-- plus the deadline of the actuator (303ms).
-- The worst-case end-to-end latency for this system on an asynchro-
nous execution platform is the
-- sum of computational latency (deadline of predecessor) rounded up
to the next multiple of
-- the periods of the three processing steps plus the actuator period
(sampling latencies)
-- plus the deadline of the predecessor of the sampler (sensor, three
steps)
-- plus the deadline of the actuator (415 ms).
system implementation application.allperiodicsampled
subcomponents
  sense: device sensor.periodic;
  actuate: device actuator.periodic;
  compute1: process Pstep1.periodic;
  compute2: process Pstep2.periodic;
  compute3: process Pstep3.periodic;
connections
  senseconn: event data port sense.outed -> compute1.ined;
  compute12: event data port compute1.outed -> compute2.ined;
  compute23: event data port compute2.outed -> compute3.ined;
  actuateconn: event data port compute3.outed -> actuate.ined;
  bus access db -> sense.devbus;
```

```
  bus access db -> actuate.devbus;
flows
  etelatency: end to end flow sense.flow1 -> senseconn -> com-
pute1.flow1
            -> compute12 -> compute2.flow1 -> compute23 -> com-
pute3.flow1
            -> actuateconn -> actuate.flow1 { latency => 303 ms;};
end application.allperiodicsampled;



-- this application configuration has all processing steps as well as
the actuator as aperiodic tasks.
-- The sensor can be periodic or aperiodic with the same result in
latency
-- The worst-case end-to-end latency for this system on a synchronous
or asynchronous execution platform is the
-- sum of the deadlines of the three processing steps plus the actua-
tor deadline and sensor deadline (computational latency) (165 ms).

system implementation application.alldatadriven
subcomponents
  sense: device sensor.periodic;
  actuate: device actuator.aperiodic;
  compute1: process Pstep1.aperiodic;
  compute2: process Pstep2.aperiodic;
  compute3: process Pstep3.aperiodic;
connections
  senseconn: event data port sense.outed -> compute1.ined;
  compute12: event data port compute1.outed -> compute2.ined;
  compute23: event data port compute2.outed -> compute3.ined;
  actuateconn: event data port compute3.outed -> actuate.ined;
  bus access db -> sense.devbus;
  bus access db -> actuate.devbus;
flows
  etelatency: end to end flow sense.flow1 -> senseconn -> com-
pute1.flow1
            -> compute12 -> compute2.flow1 -> compute23 -> com-
pute3.flow1
            -> actuateconn -> actuate.flow1 { latency => 165 ms;};
end application.alldatadriven;



-- hardware platforms: single processor, dual processor
processor singleCPU
features
  db: requires bus access devicebus;
  pb: requires bus access cpubus;
end singleCPU;

processor implementation singleCPU.basic
end singleCPU.basic;
```

```
bus cpubus
end cpubus;

bus implementation cpubus.basic
end cpubus.basic;

bus devicebus
end devicebus;

bus implementation devicebus.basic
end devicebus.basic;

system hardwareplatform
features
  db: provides bus access devicebus.basic;
end hardwareplatform;

system implementation hardwareplatform.single
subcomponents
  cpu1: processor singleCPU.basic;
  db1: bus devicebus.basic;
connections
  bus access db1 -> cpu1.db;
  bus access db1 -> db;
end hardwareplatform.single;

system implementation hardwareplatform.dual
subcomponents
  cpu1: processor singleCPU.basic;
  cpu2: processor singleCPU.basic;
  db1: bus devicebus.basic;
  cpubus1: bus cpubus.basic;
connections
  bus access db1 -> cpu1.db;
  bus access db1 -> cpu2.db;
  bus access db1 -> db;
end hardwareplatform.dual;


-- system configurations: hardware and application

system topsystem
end topsystem;

-- first all single processor configurations

system implementation topsystem.allperiodicsampled
subcomponents
  app: system application.allperiodicsampled;
  hw: system hardwareplatform.single;
connections
  dveconn: bus access hw.db -> app.db;
```

```
properties
  Actual_Processor_Binding => reference hw.cpu1 applies to app;
end topsystem.allperiodicsampled;

system implementation topsystem.alldatadriven
subcomponents
  app: system application.alldatadriven;
  hw: system hardwareplatform.single;
connections
  dveconn: bus access hw.db -> app.db;
properties
  Actual_Processor_Binding => reference hw.cpu1 applies to app;
end topsystem.alldatadriven;

-- the same application systems can be configured with a two proces-
sor system
-- we are showing one configuration where the second step is located
on a second processor
-- in this case the end-to-end latency is increased by any communica-
tion latency between the two processors across the bus
system implementation topsystem.distributedalldatadriven
subcomponents
  app: system application.alldatadriven;
  hw: system hardwareplatform.dual;
connections
  dveconn: bus access hw.db -> app.db;
properties
  Actual_Processor_Binding => reference hw.cpu1 applies to
app.compute1;
  Actual_Processor_Binding => reference hw.cpu2 applies to
app.compute2;
  Actual_Processor_Binding => reference hw.cpu1 applies to
app.compute3;
end topsystem.distributedalldatadriven;
```

Sampled Processing with Data Ports

```
-- this file contains a model that illustrates end-to-end latency due
to sampled processing
-- sampled processing occurs through data port communication.
-- the example is a signal flow from a sensor through three process-
ing steps to an actuator.
-- The first and third processing steps operate at twice the rate of
the second step.
-- The steps have a compute execution time that can vary between the
specified ranges.
-- The sensor device is the originator of the signal stream.
-- The sensors operate under two scenarios:
-- 1) the sensor periodically probes the environment, i.e., executes
periodically.
-- 2) the sensor reading is triggered by some physical event that oc-
curs randomly with a maximum rate.
-- The sampling latency is affected by whether the system operates
with respect to a global clock (synchronous system) or independent
clock (asynchronous system).
-- The models below are set up to execute under a synchronous and an
asynchronous system.


data timedata
end timedata;

-- the processing steps are defined as threads inside processes.
-- this allows them to be distributed onto different processors or
execute on the same processor.
-- The threads are periodic threads that use immediate and delayed
data port connections.
-- in other words, communication is guaranteed to always be mid-frame
or phase-delayed.
-- This controls the amount of jitter in end-to-end latency.
--
-- In a separate model we will describe the same architecture that
samples the data stream non-deterministically.


-- Step1 executes at a rate of 20 Hz and has a deadline or maximum
latency of 45 ms.
thread step1
features
  ined: in data port timedata;
  outed: out data port timedata;
flows
  flow1: flow path ined -> outed { latency => 45 ms;};
properties
      Dispatch_Protocol => Periodic;
  period => 50 ms;
  deadline => 45 ms;
  Compute_Execution_Time => 6 ms .. 10 ms;
end step1;
```

```
thread implementation step1.periodic
flows
  flow1: flow path ined -> outed;
end step1.periodic;


thread step2
features
  ined: in data port timedata;
  outed: out data port timedata;
flows
  flow1: flow path ined -> outed { latency => 70 ms;};
properties
  Dispatch_Protocol => Periodic;
  period => 100 ms;
  deadline => 70 ms;
  Compute_Execution_Time => 15 ms .. 23 ms;
End step2;

thread implementation step2.periodic
flows
  flow1: flow path ined -> outed;
end step2.periodic;


thread step3
features
  ined: in data port timedata;
  outed: out data port timedata;
flows
  flow1: flow path ined -> outed { latency => 45 ms;};
properties
  Dispatch_Protocol => Periodic;
  period => 50 ms;
  deadline => 45 ms;
  Compute_Execution_Time => 6 ms .. 10 ms;
End step3;

thread implementation step3.periodic
flows
  flow1: flow path ined -> outed;
end step3.periodic;

-- at the beginning of each dispatch the sensor device reads the
clock
-- and passes it as the value of its output
device sensor
features
  outed: out data port timedata;
  devbus: requires bus access devicebus;
flows
  flow1: flow source outed { latency => 2 ms;};
```

```
properties
  period => 50 ms;
  deadline => 2 ms;
  Compute_Execution_Time => 1 ms .. 2 ms;
end sensor;

-- sensor periodically senses the physical environment
device implementation sensor.periodic
flows
  flow1: flow source outed;
properties
  Device_Dispatch_Protocol => Periodic;
end sensor.periodic;

-- sensor detects an in the physical environment
-- the occurs randomly with a maximum rate of the period
device implementation sensor.aperiodic
flows
  flow1: flow source outed;
properties
  Device_Dispatch_Protocol => Aperiodic;
end sensor.aperiodic;

-- The actuator will read the clock
-- and log the difference to the received data (sensor clock time) as
its last action
device actuator
features
  ined: in data port timedata;
  devbus: requires bus access devicebus;
flows
  flow1: flow sink ined { latency => 3 ms;};
properties
  period => 50 ms;
  deadline => 3 ms;
  Compute_Execution_Time => 1 ms .. 3 ms;
end actuator;

-- output is sampled. This reduces the latency jitter.
device implementation actuator.periodic
flows
  flow1: flow sink ined;
properties
  Device_Dispatch_Protocol => Periodic;
end actuator.periodic;

-- arrival of data causes actuator to become active.
-- This reduces end-to-end latency at the expense of increased lit-
ter.
device implementation actuator.aperiodic
flows
```

```
  flow1: flow sink ined;
properties
  Device_Dispatch_Protocol => Aperiodic;
end actuator.aperiodic;

process Pstep1
features
  ined: in data port timedata;
  outed: out data port timedata;
flows
  flow1: flow path ined -> outed;
end Pstep1;

process implementation Pstep1.periodic
subcomponents
  Tstep1: thread Step1.periodic;
connections
  cin: data port ined -> Tstep1.ined;
  cout: data port Tstep1.outed -> outed;
flows
  flow1: flow path ined -> cin -> Tstep1.flow1 -> cout -> outed;
end Pstep1.periodic;


process Pstep2
features
  ined: in data port timedata;
  outed: out data port timedata;
flows
  flow1: flow path ined -> outed;
end Pstep2;

process implementation Pstep2.periodic
subcomponents
  Tstep2: thread Step2.periodic;
connections
  cin: data port ined -> Tstep2.ined;
  cout: data port Tstep2.outed -> outed;
flows
  flow1: flow path ined -> cin -> Tstep2.flow1 -> cout -> outed;
end Pstep2.periodic;


process Pstep3
features
  ined: in data port timedata;
  outed: out data port timedata;
flows
  flow1: flow path ined -> outed;
end Pstep3;

process implementation Pstep3.periodic
```

```
subcomponents
  Tstep3: thread Step3.periodic;
connections
  cin: data port ined -> Tstep3.ined;
  cout: data port Tstep3.outed -> outed;
flows
  flow1: flow path ined -> cin -> Tstep3.flow1 -> cout -> outed;
end Pstep3.periodic;


system application
features
  db: requires bus access devicebus;
end application;

-- this application configuration has all processing steps as well as
the sensor and actuator as periodic tasks.
-- The connections are delayed connections to allow for deterministic
sampling at each step.
-- The worst-case end-to-end latency for this system on a synchronous
execution platform is the
-- sum of computational latency (deadline of predecessor) rounded up
to the next multiple of
-- the periods of the three processing steps plus the actuator period
(sampling latencies)
-- plus the deadline of the actuator (303ms).
-- The worst-case end-to-end latency for this system on an asynchro-
nous execution platform is the
-- sum of computational latency (deadline of predecessor) rounded up
to the next multiple of
-- the periods of the three processing steps plus the actuator period
(sampling latencies)
-- plus the deadline of the predecessor of the sampler (sensor, three
steps)
-- plus the deadline of the actuator (415 ms).
system implementation application.allperiodicdelayed
subcomponents
  sense: device sensor.periodic;
  actuate: device actuator.periodic;
  compute1: process Pstep1.periodic;
  compute2: process Pstep2.periodic;
  compute3: process Pstep3.periodic;
connections
  senseconn: data port sense.outed ->> compute1.ined;
  compute12: data port compute1.outed ->> compute2.ined;
  compute23: data port compute2.outed ->> compute3.ined;
  actuateconn: data port compute3.outed ->> actuate.ined;
  bus access db -> sense.devbus;
  bus access db -> actuate.devbus;
flows
  etelatency: end to end flow sense.flow1 -> senseconn -> com-
pute1.flow1
            -> compute12 -> compute2.flow1 -> compute23 -> com-
pute3.flow1
```

```
            -> actuateconn -> actuate.flow1 { latency => 303 ms;};
end application.allperiodicdelayed;


-- this application configuration has all processing steps as well as
the actuator as periodic tasks.
-- The sensor operates periodically (aperiodic sensor action in-
creases the latency by the deadline of the third step).
-- The connections are immediate connections to allow for determinis-
tic processing within the same frame.
-- The actuator connection is delayed to allow for phase delayed sam-
pling to minimize latency jitter for the actuation.
-- The worst-case end-to-end latency for this system on a synchronous
execution platform is the
-- the deadline of the last processing step rounded up to the actua-
tor period (sampling latency) and actuator deadlines (computational
latency) (53 ms).
-- In the asynchronous case the latency increases by the deadline of
the third step, since the actuator samples independently.


system implementation application.allimmediate
subcomponents
  sense: device sensor.periodic;
  actuate: device actuator.periodic;
  compute1: process Pstep1.periodic;
  compute2: process Pstep2.periodic;
  compute3: process Pstep3.periodic;
connections
  senseconn: data port sense.outed -> compute1.ined;
  compute12: data port compute1.outed -> compute2.ined;
  compute23: data port compute2.outed -> compute3.ined;
  actuateconn: data port compute3.outed ->> actuate.ined;
  bus access db -> sense.devbus;
  bus access db -> actuate.devbus;
flows
  etelatency: end to end flow sense.flow1 -> senseconn -> com-
pute1.flow1
            -> compute12 -> compute2.flow1 -> compute23 -> com-
pute3.flow1
            -> actuateconn -> actuate.flow1 { latency => 53 ms;};
end application.allimmediate;


-- this application configuration has all processing steps as well as
the actuator as periodic tasks.
-- The sensor operates periodically (aperiodic sensor action in-
creases the latency by the deadline of the second step).
-- The connections are immediate to the first step, delayed for the
second step to force phase-delayed sampling, immediate to the third
step, and delayed to the actuator.
-- In other words, there are two sampling steps, the computation of
step2, and the actuator action.
-- The worst-case end-to-end latency for this system on a synchronous
execution platform is the
```

```
-- the deadline of the first processing step rounded up to the second
step period, plus the third step deadline rounded up to the actuator
period (sampling latency) plus actuator deadlines (computational la-
tency) (153 ms).
-- In the asynchronous case the latency increases by the deadlines of
the first and third steps.

system implementation application.twosamplesteps
subcomponents
  sense: device sensor.periodic;
  actuate: device actuator.periodic;
  compute1: process Pstep1.periodic;
  compute2: process Pstep2.periodic;
  compute3: process Pstep3.periodic;
connections
  senseconn: data port sense.outed -> compute1.ined;
  compute12: data port compute1.outed ->> compute2.ined;
  compute23: data port compute2.outed -> compute3.ined;
  actuateconn: data port compute3.outed ->> actuate.ined;
  bus access db -> sense.devbus;
  bus access db -> actuate.devbus;
flows
  etelatency: end to end flow sense.flow1 -> senseconn -> com-
pute1.flow1
            -> compute12 -> compute2.flow1 -> compute23 -> com-
pute3.flow1
            -> actuateconn -> actuate.flow1 { latency => 153 ms;};
end application.twosamplesteps;

-- hardware platforms: single processor, dual processor
processor singleCPU
features
  db: requires bus access devicebus;
  pb: requires bus access cpubus;
end singleCPU;

processor implementation singleCPU.basic
end singleCPU.basic;

bus cpubus
end cpubus;

bus implementation cpubus.basic
end cpubus.basic;

bus devicebus
end devicebus;

bus implementation devicebus.basic
end devicebus.basic;

system hardwareplatform
features
```

```
  db: provides bus access devicebus.basic;
end hardwareplatform;

system implementation hardwareplatform.single
subcomponents
  cpu1: processor singleCPU.basic;
  db1: bus devicebus.basic;
connections
  bus access db1 -> cpu1.db;
  bus access db1 -> db;
end hardwareplatform.single;

system implementation hardwareplatform.dual
subcomponents
  cpu1: processor singleCPU.basic;
  cpu2: processor singleCPU.basic;
  db1: bus devicebus.basic;
  cpubus1: bus cpubus.basic;
connections
  bus access db1 -> cpu1.db;
  bus access db1 -> cpu2.db;
  bus access db1 -> db;
end hardwareplatform.dual;


-- system configurations: hardware and application

system topsystem
end topsystem;

-- first all single processor configurations

system implementation topsystem.allperiodicdelayed
subcomponents
  app: system application.allperiodicdelayed;
  hw: system hardwareplatform.single;
connections
  dveconn: bus access hw.db -> app.db;
properties
  Actual_Processor_Binding => reference hw.cpu1 applies to app;
end topsystem.allperiodicdelayed;

system implementation topsystem.allimmediate
subcomponents
  app: system application.allimmediate;
  hw: system hardwareplatform.single;
connections
  dveconn: bus access hw.db -> app.db;
properties
  Actual_Processor_Binding => reference hw.cpu1 applies to app;
end topsystem.allimmediate;
```

```
system implementation topsystem.twosamplesteps
subcomponents
  app: system application.twosamplesteps;
  hw: system hardwareplatform.single;
connections
  dveconn: bus access hw.db -> app.db;
properties
  Actual_Processor_Binding => reference hw.cpu1 applies to app;
end topsystem.twosamplesteps;

-- the same application systems can be configured with a two proces-
sor system
-- we are showing one configuration where the second step is located
on a second processor


system implementation topsystem.distributedallperiodicdelayed
subcomponents
  app: system application.allperiodicdelayed;
  hw: system hardwareplatform.dual;
connections
  dveconn: bus access hw.db -> app.db;
properties
  Actual_Processor_Binding => reference hw.cpu1 applies to
app.compute1;
  Actual_Processor_Binding => reference hw.cpu2 applies to
app.compute2;
  Actual_Processor_Binding => reference hw.cpu1 applies to
app.compute3;
end topsystem.distributedallperiodicdelayed;
```

References

URLs are valid as of the publication date of this document.

[ARINC653 2003]
ARINC. *ARINC Specification 653P1-2. 653P1-2 Avionics Application Software Standard Interface, Part 1 - Required Services* (2003)
https://www.arinc.com/cf/store/catalog_detail.cfm?item_id=632.

[Cervin 2006]
Cervin, A.; Årzén, K.-E.; & Henriksson, D. "Control Loop Timing Analysis Using TrueTime and Jitterbug," 1194–1199. *Proceedings of the 2006 IEEE Conference on Computer Aided Control Systems Design (CACSD).* Munich, Germany, October 4–6, 2006.
http://ieeexplore.ieee.org/iel5/4064705/4064706/04064765.pdf?tp=&arnumber=4064765&isnumber=4064706

[Feiler 2006]
Feiler, Peter H.; Gluch, David P.; & Hudak, John. *The Architecture Analysis & Design Language (AADL): An Introduction* (CMU/SEI-2006-TN-011). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2006.

[Hansson et al. 2007]
Hansson J., Feiler, P., Greenhouse, A., Hudak, J., and Wrage, L.. *Impact of Architecture Concurrency on Performance Engineering* (CMU/SEI-2007-TN-048). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2007.

[SAE AS5506 2004]
Society of Automotive Engineers. *SAE Standards: AS5506, Architecture Analysis & Design Language (AADL)*, November 2004.
http://www.sae.org/servlets/productDetail?PROD_TYP=STD&PROD_CD=AS5506

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	June 2007	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Flow Latency Analysis with AADL	FA8721-05-C-0003

6. AUTHOR(S)
Peter Feiler

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2007-TN-010

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Control system components are sensitive to the end-to-end latency and age of signal data as well as their variation (jitter). Typically, components assume data latency, age, and variation to be within a given range. Different runtime configurations (i.e., different sampling and data driven signal processing pipelines, different communication mechanisms, partitioned architectures, and globally synchronous versus asynchronous hardware) affect these measures.

This technical note describes the flow specification notation in the Architecture Analysis and Design Language (AADL) and introduces a latency analysis framework. This framework is able to determine the end-to-end latency and age of a signal stream data as well as their variation (jitter). The latency calculations are illustrated in the context of an example AADL model. The report illustrates how this latency analysis capability can be used to determine worst-case end-to-end latency on system models of different fidelity and can take into account partitioned architectures. The report summarizes the worst-case end-to-end flow latency analysis capabilities that are provided by the Flow Latency Analysis plug-in for the Open Source AADL Tool Environment (OSATE).

14. SUBJECT TERMS	15. NUMBER OF PAGES
Architecture, AADL, flow latency, end-to-end analysis	59

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18
298-102