Impact of Runtime Architectures on Control System Stability

P Feiler, J Hansson

Impact of Runtime Architectures on Control System Stability

P. Feiler, J. Hansson

Software Engineering Institute, Pittsburgh, PA

Abstract: Control systems are sensitive to the end-to-end latency and age of signal data. Control engineers develop their control system model under certain latency and age assumptions and deviation from these assumptions can lead to controller instability. In this paper we discuss how choices in the runtime architecture of the embedded software system can affect latency and introduce unexpected latency jitter. We propose the use of AADL as a basis for a flexible framework that support co-design of control systems by control engineers and by embedded software engineers through quantitative analysis.

Keywords: Embedded system, control system stability, end-to-end latency, latency jitter

1. Introduction

Control systems are highly time sensitive. They are developed by control engineers through a process of model validation. The physical plant and the desired control algorithm are represented by continuous time models to capture the physics of the system. They are then mapped into discrete time models to take into account digital processing by software. Finally, they control components are translated into source code. These models may exist in different levels of fidelity, and they are validated through simulation and model checking and source code execution in a simulated environment and with hardware in the loop.

During this development process the control algorithms are calibrated to the physical system characteristics. Reflected in the calibration are assumptions regarding the end-to-end latency of the signal stream from the sensor and to the actuator, as well as the latency jitter and the age of signal data. One of these assumptions is that the runtime system exhibits deterministic execution and communication behavior.

Control system applications are components of an embedded software system. These components interact in the context of specific runtime architectures utilizing a variety of communication and scheduling mechanisms and policies. Similarly, the resulting runtime architecture deployed and distributed on different hardware platforms. These choices are often made by software engineers with limited awareness of their impact on latency, latency jitter, and age of signal streams are affected. Without managing the impact by ensuring the latency assumptions made by control engineers in the runtime architecture the stability of the controllers is impacted without recalibration.

This separation of concerns between control and computation has traditionally been addressed by first developing the control algorithm, and then configuring the resulting software components into an embedded application. This approach was feasible while control systems were physically separate components with their dedicated processors. As embedded systems increasingly use a common compute hardware platform that is shared among software implementations of control components, and embedded systems increasingly require control components to interact to provide desired functionality, the work of embedded software engineers must go hand in hand with that of control engineers. In this co-design approach embedded software engineers must be able to quantify latency and latency jitter contributions by choices in the runtime system implementation and ensure latency assumptions of control algorithms where critical. Similarly, control engineers must be able to quantify the robustness of their control algorithms with respect to variation in latency in order to allow for more flexibility in the runtime architecture and better utilization of the compute hardware [1].

In this paper we utilize the international industry standard notation SAE AADL [2] to gain a better understanding of the impact of runtime architecture choices on controller stability. AADL has been designed to characterize embedded software applications, and deployment of compute hardware, and their interface with the physical environment. Its rich semantics with respect to task execution and communication between components allows us to model the signal flow processing architecture as a flow-based architecture. We will use AADL to model the execution and communication timing characteristics assumed by control engineers are they develop their control design. We will also use the AADL to characterize various aspects of the runtime architecture of a control system implementation and identify contributors to end-to-end latency and to latency jitter. This analysis will highlight the importance of determinism to manage latency jitter to maintain the stability of controllers under jitter assumptions established by control engineers.

In this paper we will first characterize the latency and latency jitter assumptions made by control engineers

as they develop their design independent of software implementation considerations. Then we will examine contributors to latency and latency jitter in the runtime architecture and identify a systematic way of determining such impact. Finally, we identify the need for a flexible analytic framework for determining end-to-end latency and latency jitter that embedded software engineers and control engineers to make informed design choices through model-based analysis early and throughout the development process in a co-design design setting.

2. A Control Engineering Perspective

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.

The control systems are initially expressed as continuous time models in differential equations in order to capture the physical characteristics and behaviors of the physical systems. These models are then mapped into discrete time models to reflect sampled control in a computer-based control system implementation [3]. Simulink is an example of a commercial modeling environment for control engineers that supports both continuous time and discrete time modeling. Such modeling environments assume a computational model whose execution behavior is implemented in their simulation engine.

In the next sections we will examine this assumed model of computation and communication, model its essence in AADL, and discuss its sensitivity to latency variation and jitter.

2.1 A Model of Computation and Communication

The computational model consists of computational components that execute periodically with an input-compute-output (ICO) behavior. This model provides data consistency during computation in that the input to the computation is determined in the input phase and is not affected by newly arriving output during the time of computation. Similarly, output is made available to successors during the output phase upon completion of computation. Data consistency of communication is established through a port-based communication model with atomic send/receive (write/read) operations. Such port automata represent an algebra of concurrent processes [4].

In a modeling environment such as Simulink this computational model is represented by blocks representing components, pins representing ports, and connections representing communication of data through variables from the output of one component to the input of another component. Components execute at a specified rate. Different components may execute at different rates, resulting in over- and under-sampling of the predecessor output. The simulation executes the components sequentially in discrete time frames at the specified rates. Within a frame the execution order of the components is fixed and can be specified by the modeler.

The execution order of the components determines the send/receive order of the input and output operations, i.e., determines whether the data is received by the receiver within the same frame or at the next frame. If the recipient executes after the sender the communication is mid-frame, and if the recipient executes before the sender the communication is phase-delayed.

When communicating components execute at different rates, rate transformation blocks, representing zero-order hold and unit-delay, effectively perform double buffering of the result in order to assure data consistency of computation while performing mid-frame and phase-delayed communication.

Since the execution order is always the same, data is always communicated deterministically mid-frame or phase-delayed between components. The end-to-end latency of a data stream determined by the initial sampling of the sensor readings, the processing time of sequences of components with mid-frame communication. This cumulative processing latency is then sampled by phase-delayed communication with the cumulative processing latency rounded up to the next frame. This provides an effective way of managing execution time jitter, i.e., variation in actual execution time of individual components. For example, the control system may apply a filter to the sensor data before computing the output to the actuator. These two tasks are shown as T11 and T12 in Figure 1. By passing the output phase-delayed to task T2, T2 will feed data to the actuator at the beginning of the next frame independent of any variation in execution time to perform the filtering or control computation. Often the task providing data to the actuator executes at a multiple rate of the control algorithm and deterministically oversamples the controller output.

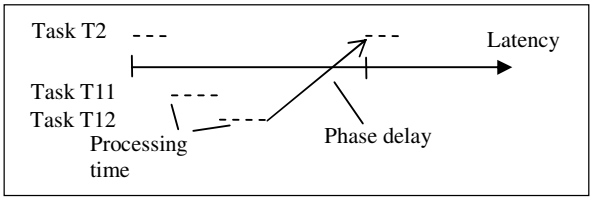

Figure 1: Processing Time and Sampling Latency

2.2 An AADL Model of the Control System

AADL supports modeling of periodic and event or data-driven tasks (AADL threads). These threads

can have data ports, i.e., unqueued ports that communicate state data, as well as ports for communicating events and messages. Data ports make the most recently recent data value available to a thread at thread dispatch time. During the execution of the thread this data value is not affected by any newly received data. The output of data ports is made available to other threads at execution completion. In other words, the AADL thread model supports the ICO behavior desirable for control systems.

For periodic threads with data ports users can specify immediate and delayed port connections. The timing semantics correspond to mid-frame and phase-delayed communication. In the case of an immediate connection the execution of the recipient thread is suspended until the sending thread completes its execution and makes its output available to the recipient. In the case of a delayed connection the output of the sending thread is not transferred until the sending thread's deadline, typically the end of the period. In other words, its output is not available to the recipient until the next frame. These communication timing semantic assure deterministic over- and under-sampling when a data stream is processed. The timing characteristics of immediate and delayed connections are shown in Figure 2.

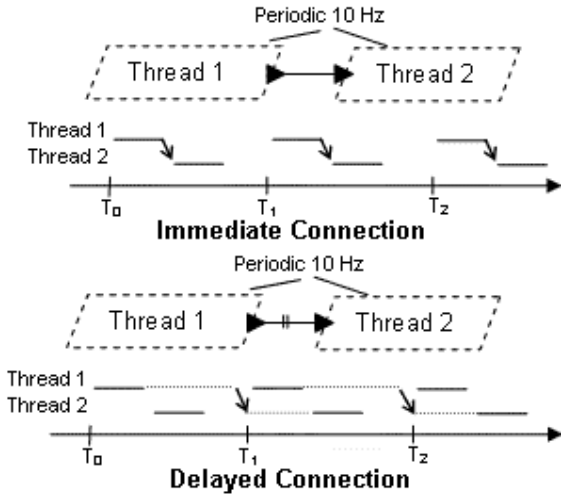

Figure 2 Immediate and Delayed Connections

This allows us to describe the expected execution and communication timing behavior of a control system simply as a sensor device that samples the physical environment at a given period. Its output is fed to a filter thread via immediate connection. This filter thread and the control computation thread, which is connected via immediate connection, execute at twice the period of the sensor device. The output of the control computation is fed to an actuator device via delayed connection. The actuator device is specified to execute at half the period of the control computation.

AADL also allows us to specify an end-to-end flow. This is done by first specifying a flow source for the sensor, flow paths for the filter and control thread, and a flow sink for the actuator. These flow specifications provide an external specification of information flow path through an AADL component from an incoming port to an outgoing port (in case of a flow source starting from within a component, and a flow sink ending within a component). If a component consists of subcomponents, this flow is elaborated within the component implementation as a flow through these subcomponents. An end-to-end flow is specified by starting with the flow source of a component, in our case the sensor device, to follow the connection to the filter thread, following its flow path and the connection to the control thread, the control thread flow path and the connection to the actuator device flow sink. The end-to-end flow and the flow specifications can have AADL properties, in our case the specification of the desired end-to-end flow latency attached to the end-to-end flow specification. The latency within a sensor or actuator can be attached to its flow source or flow sink specification. The latency contributed by a thread can be determined from its period, deadline, and worst-case execution time and the type of data port connection [5].

AADL also supports specification of sampling port connections as well as data driven processing with arrival of data triggering the execution of threads. In addition, AADL supports modeling of shared access to data components by threads with write and read access determining the information flow.

2.3 Non-determinism in Computation and Communication

Control algorithms are sensitive to latency and sampling jitter as well as variation in age of the data. 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 value being processed may be larger than the end-to-end latency.

Cervin et.al. [1] illustrate how sampling jitter and latency jitter affect the stability of controllers. They also show that jitter varies according to the scheduling algorithm used for executing a task set. The standard task model with a single assigned priority and input and output performed as part of this task performs worse than a task model in which input and output is managed separately at higher priority, thus, making task interaction more

deterministic. Sampling and latency jitter as well as variation in data age is perceived by the control algorithm as increased noise in the data causing the control algorithm to become less stable.

In the next section we examine how latency and age of data streams are affected by choices made by software engineers as they integrate control components into a set of communicating software tasks that share processors, execute concurrently on different processors, and communicate over high-speed or slow communication channels. Those choices may introduce unintended non-determinism, thus, increase latency jitter [6].

3. Embedded Software Engineering Perspective

When implementing an embedded software system software engineers make a number of decisions regarding the runtime system of the application. Algorithms may vary in execution time, tasks may be scheduled on a static time line or may execute pre-emptively to improve processor utilization. Tasks may communicate through shared data variables or use a send/receive communication paradigm. Multiple tasks executing at the same rate may be executed in the same operating system thread in order to reduce the number of context switches. The embedded system may be ported to a partitioned architecture in order to improve configurability and deployment options of the embedded system. Different communication protocols can be used for communication across processors. Different processor in a distributed system operate asynchronously on different clocks. All of these considerations can have an impact on the end-to-end latency and its jitter. In the next sections we discuss the impact of these latency contributors.

3.1 Execution Time Variation

Latency jitter is due to variation in actual execution time. Different data values may require different amounts of computation. Latency jitter is also due to the use of preemptive scheduling techniques in order to increase the utilization of processors. Preemptive scheduling causes one task to be preempted to allow a higher priority task to complete, delaying its completion by varying amounts of time. As pointed out earlier, such execution time variation of a data stream with can be masked by a sampling task ass long as the jitter does not exceed the sampling rate.

Variation in execution time is recorded in AADL models as a time range. Similarly, AADL processors have a property that indicates the type of scheduling protocol, i.e., it can identify whether preemptive scheduling is being performed.

3.2 Non-Deterministic Communication

Preemptive scheduling of tasks as well as concurrent execution of tasks on different processors, e.g., different cores one a multi-core processor, can have a greater effect on jitter. The execution of send/receive (e.g., write/read to variables) to accomplish the communication, when executed as part of the application code, can lead to a non-deterministic send/receive order [7]. This leads to frame-level latency jitter. Let us illustrate with an example shown in Figure 3.

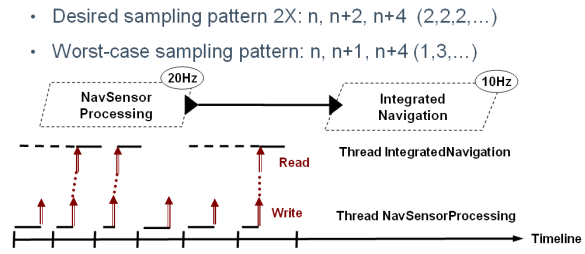

Figure 3 Frame-level Latency Jitter

Mid-frame and phase-delayed communication semantics guarantee that a task deterministically samples a data stream. For example, if the rates of two communicating tasks are harmonic, i.e., the sending task rate is twice the rate of the receiver, then the receiver reads every second element in the data stream written by the sending task.

However, if the write and read order is not guaranteed to be deterministic, then the receiving tasks may read two successive elements in the data stream and then skip two elements to read the third. The effect is that sampling of the data stream may vary by as much as two frames.

In AADL, deterministic data port communication can be explicitly modeled through immediate and delayed connections. In addition, AADL supports modeling of sampled port connection as well as explicit modeling of communication through shared data components.

3.3 Rate Group Optimization

Rate group optimization is the process of mapping tasks that have the same execution rate on a specific processor into a single operating system (OS) thread of that rate. The benefit of rate group optimization is that context switch time between tasks of the same rate is greatly reduced because the operating system thread executes these tasks on a static time by calling them as subprograms.

Rate group optimization affects the task execution order within a frame as it places tasks of the same rate into groups and executes them in order within each group.

A task sequence with mid-frame communication with tasks of different rates has an expected execution order of these tasks. The rate group optimization may result in a task execution order that is in conflict with the desired order for the task sequence.

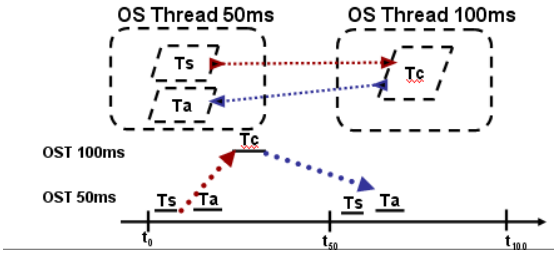

Figure 4 Phase Delay due to Rate Group Optimization

Let us illustrate with an example shown in Figure 4. Let us assume a sequence of three tasks with mid-frame communication. Task Ts and Ta are higher rate tasks (50 ms), while Tc is a lower rate task (100ms). Rate group optimization will place Ts and Ta in a high rate group G1, and Tc in a low rate group G2. If G1 is executed before G2, e.g., if rate group priority is assigned based on rates, the constraint Tc->Ta is violated, and if G2 is executed before G1 the constraint Ts->Tc is violated. In other words, the rate group optimization forces one of the mid-frame communication steps to become a phase-delayed communication step.

The general rule to be checked is whether there is an immediate connection between threads in different rate groups and the execution order of those rate groups cannot be guaranteed to be the same as that implied by the immediate connection. A corollary to this rule is that if there exists an immediate connection from a thread in one rate group to a thread in a second rate group, and there exists a second immediate connection from a thread in the second rate group to a thread in the first rate group the immediate connection timing semantics cannot be guaranteed.

When the immediate connection semantics cannot be guaranteed one of two things happens to the end-to-end latency: the end-to-end latency will increase if the execution order of the rate groups deterministic; or latency jitter will occur due to preemption or concurrent execution of rate group threads.

AADL supports modeling of rate group optimization at several levels of abstraction. For example, logical tasks can be modeled by AADL threads. An AADL property can be introduced to define the mapping into a rate group for each AADL thread. The actual implementation of the rate group as an operating system thread is then derived (generation) from this model. Other modeling options to more explicitly represent the rate group include the use of virtual processors of AADL V2 [9].

3.4 Partitioned Architectures

Partitioning is used to support integrated modular avionics (IMA). A partition provides a virtual processor that ensures space partitioning through address space protection and time partitioning by ensuring its processor allocation is not exceeded. Within partitions multiple threads may execute and each partition may implement its own policy for scheduling its threads. All interactions between partitions is accomplished through port-based communication. This allows embedded applications to be modularized and configured in different ways to run on a common computing platform.

The ARINC 653 standard for embedded avionics systems [8] specifies that all inter-partition communication must be phase-delayed if deterministic communication behavior is desired (see **Figure 5**). This makes the behavior of the application insensitive of the partition execution order or partitions executing concurrently. However, this may require double buffering in order to assure the correct data to be available to the recipient. As we place an embedded application into a partitioned architecture and the end-to-end flow spans multiple partitions, then the end-to-end latency is increased by the phase-delayed communication across partition boundaries.

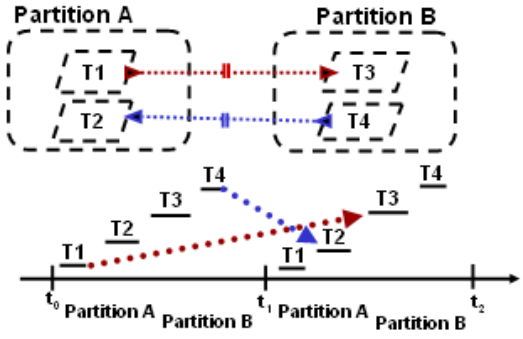

Figure 5 Phase Delayed Inter-partition Communication

The phase-delayed inter-partition communication model can have unexpected side-effects on existing legacy applications that implement an ICO model. A common way of implementing this model is to have a high priority task (*Periodic I/O*) take the output from a shared data area and send it to other control subsystems, as well as place the input from other subsystems into the shared data area (see **Figure 6**). Note that this has the effect of phase-delayed communication. When ported into a partitioned architecture, the partition communication mechanism will phase delay the cross-partition data traffic, but

the data it receives from the application, i.e., its *Periodic I/O* task, is already delayed. This will result in doubling of the latency contributed by communication across partitions.

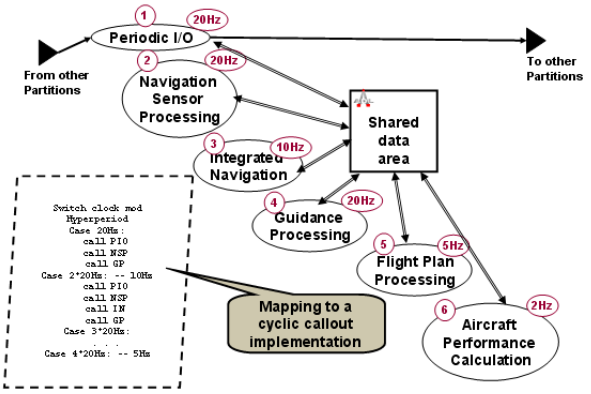

Figure 6 Legacy Cyclic Executive Implementation

If the partitioned architecture does not ensure phase delayed communication, e.g., by application send/receive calls without the communication protocol performing double-buffering to delay the receipt, we get two effects on the end-to-end latency. Let us first examine partitions on the same processor. In this case communication from the first partition to the second partition is mid-frame, while communication from the second partition to the first partition is phase-delayed. If we change the statically determined execution order of partitions, then the immediate connection becomes delayed and the delayed connection becomes immediate – resulting in a change in the flow latency through either connection.

If the partitions execute on different processors, then the send/receive order may be non-deterministic resulting in frame-level jitter due to concurrent execution. If the execution order of partitions is changed on one of the processors and the processors clocks are synchronized, one partition may always be executing before the other, i.e., the execution order may become predictable with communication in one direction immediate and the other delayed.

In summary, in a partitioned architecture we can isolated the embedded application from non-determinism, or we can determine whether a control application needs to accommodate latency jitter or be recalibrated for a change in latency.

Partitions can be modelled in AADL through AADL processes. They represent space partitions in their default semantics. Partition-specific properties can be introduced to characterize the partition execution rate and the scheduling protocol supported by the partition for executing its threads. In AADL V2 [9] we can use the virtual processor concept to more explicitly represent the partition as an entity that provides time partitioning and scheduling of threads.

3.5 Communication Protocols

Communication protocols contribute to end-to-end latency as well. There is the transmission latency, i.e., the amount of time it takes to transmit the data from the source to the destination. This figure is dependent on the speed of the underlying transmission medium, and on the amount of data to be transmitted. It is the equivalent to processing latency contributed by tasks. In addition, communication protocols contribute transmission delay due to queuing or due to waiting for the transmission time slot. Examples of the latter are signal data transfer over CANBus or a time-triggered protocol. Such time-division protocols can be viewed as sampling the data stream to be transmitted, thus, contributing sampling latency.

Communication protocols are modeled as part of the AADL bus abstraction. The AADL bus has a number of properties that allow modelers to specify data transmission costs. Additional properties can be introduced to specify protocol characteristics such as guaranteed delivery, and slot assignment in a time-division protocol. In order to better support modeling of protocols independent of physical connections, AADL V2 [9] has introduced the virtual bus concept.

3.6 Globally Asynchronous Systems

In a synchronous system, task dispatches are aligned. As a result, the sampling latency can be determined by rounding the processing latency to the next multiple of the sampling rate (see also Section 2.1).

In a globally asynchronous system, the sampling latency has to be added to the processing latency to accommodate worst-case assumptions of misalignment of clocks. Furthermore, clock drift becomes evident as small changes in latency jitter. Resynchronization of clocks has the effect of a jump in latency, whose size is dependent on the degree of drift and resynchronization frequency.

The base semantics of AADL [1] are defined in terms of synchronous systems. However, the standard allows AADL properties to be used to introduce clock asynchronicity. For example, AADL processor can have properties that indicate the clock used as its time reference point. The AADL device concept can be used to represent clocks explicitly and associate clock specific properties, such as drift with respect to some universal time. The AADL V2 standard [9] will provide additional guidance and support for modeling multiple time spaces, i.e., support for modeling various forms of asynchronicity such as globally asynchronous locally synchronous (GALS) systems or physically asynchronous logically synchronous (PALS) systems.

4. A Flexible Framework for Latency Analysis

There is a need for control engineers and embedded software engineers to cooperate in a co-design setting when developing software-intensive control systems. In support of such a co-design environment there is a need for flexible framework for end-to-end latency analysis. This framework must be flexible in several dimensions. First, it must usable early and throughout the development life cycle, i.e., it must accommodate multi-fidelity modeling and analysis. This means it must support partial models of systems with few runtime architecture decisions that can produce initial insights, and that can later be refined into more complete models for further analysis. Second, the framework must be extensible to accommodate new contributors to latency and latency jitter as new runtime architectures and communication mechanisms are developed and deployed.

This analysis framework must accommodate a few principal concepts of time-sensitive data stream processing and their realization in software [6]. These include synchronous and asynchronous processing and communication, sampled and data-driven processing and communication, characterization of determinism, resource sharing delay, and predictable worst-case and best-case performance. By centering the framework around data stream characteristics we can express assumed timing characteristics from a control engineering perspective, and contributors to these characteristics due to processing tasks and communication mechanisms in the runtime architecture of the software implementation.

We suggest that AADL can be a good platform for such an analytical framework. AADL allows modelers to capture expected data stream characteristics, and its allows users to represent the runtime architecture of the embedded software system at different levels of abstraction. We have developed an initial realization of such a framework around AADL [5]. In this framework we have demonstrated how a lower bound of worst-case end-to-end latency can be quantified for models of the control system at various levels of fidelity. For example, end-to-end latency may be calculated based decisions regarding partitioning of major subsystems without any details about the tasks, and later revisited when processing tasks are known in terms of sampling, period, and communication timing, when binding decisions are made with respect to deployment on compute hardware. This framework can be extended to accommodate contributions of protocols used in the communication, effects of data stream miss rates, and choices in fault tolerance mechanisms.

This framework can accommodate worst-case and best-case latency, as well as determination of age. The impact of such timing measures of data streams on control system behaviour can also be determined analytically [3]. Commercial tools are starting to address this need for specific application domains, e.g., the automotive domain [10].

5. Conclusion

In this paper we have examined the sensitivity of control systems to implementation choices in runtime architectures for embedded software systems. For a control engineering perspective we have identified commonly assumed timing characteristics for processing a data stream in a control loop. We have identified non-determinism in sampling a data stream as a key contributor to latency jitter, which in turn causes instability in control behavior. We have then examined several runtime architecture concepts for their contributions to latency and latency jitter. We have shown that AADL can be used to characterize the control application and the runtime system that implements this control application and that the semantics associated with AADL concepts are well suited to capture the essence of the timing problem space to be the basis for an adaptable analytical framework that allows control engineers and embedded software engineers to evolve the design of a system in a co-design setting through repeated analysis of models of different fidelity.

7. References

[1] 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.

[2] Society of Automotive Engineers. “Architecture Analysis & Design Language (AADL)”, SAE Standards: AS5506, P. Feiler (ed.), November 2004.

[3] Åström, K.J.; Wittenmark, B. “Computer Controlled Systems – Theory and Design” Prentice-Hall, 1996.

[4] Steenstrup, M., Arbib, M.A., and Manes, E.G., “Port Automata and the Algebra of Concurrent Processes" Journal of Computer and System Sciences, Vol. 27, No. 1, Aug. 1983, pp.29-50.

[5] Feiler, P.; Hansson, J. “Flow Latency Analysis with the Architecture Analysis and Design Language (AADL),” Technical Note CMU/SEI-2007-TN-010, Software Engineering Institute, Dec 2007.

[6] Caspi, P.; Maler, O. “On the Implementation of Control Loops by Software,” 1574–1581. Proceedings of the 2006 IEEE Conference on

Computer Aided Control Systems Design (CACSD). Munich, Germany, October 4–6, 2006.

[7] Feiler, P. “Efficient Embedded Runtime Systems Through Port Communication Optimization” Proceedings of the 13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2008). UML&AADL 2008 Workshop. Belfast, Northern Ireland, April, 2008.

[8] ARINC. ARINC Specification 653P1-2. 653P1-2 Avionics Application Software Standard Interface, Part 1 - Required Services (2003).

[9] Society of Automotive Engineers. “Architecture Analysis & Design Language (AADL) Version 2”, SAE Standards: AS5506-2, P. Feiler (ed.), expected in Fall 2008.

[10] Symta Vision. End-to-end timing analysis for gated networks. www.symtavision.com.

8. Glossary

AADL: Architecture Analysis & Design Language

ARINC: Aeronautical Radio Inc.

CAN Bus: Controller Area Network Bus

GALS: Globally asynchronous Locally Synchronous Systems

ICO: Input-Compute-Output

IMA: Integrated Modular Avionics

PALS: Physically asynchronous Logically Synchronous Systems

SAE: Society of Automotive Engineers