Embedded System Architecture Analysis Using SAE AADL

Peter H. Feiler (SEI)
David P. Gluch (Embry-Riddle Aeronautic University/SEI)
John J. Hudak (SEI)
Bruce A. Lewis (U.S. Army AMRDEC)

June 2004

Performance-Critical Systems Initiative

Technical Note
CMU/SEI-2004-TN-005

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.

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).

Contents

List of Figures

Abstract

The emerging Society of Automotive Engineers Architecture Analysis and Design Language (AADL) standard is an architecture modeling language for real-time, fault-tolerant, scalable, embedded, multiprocessor systems. It enables the development and predictable integration of highly evolvable systems as well as analysis of existing systems. It supports early and repeated analyses of a system's architecture with respect to performance-critical properties through an extendable notation, a tool framework, and precisely defined semantics.

This report discusses the role and benefits of using the AADL in the process of analyzing an existing avionics system. The AADL is used to describe architecture patterns in the system being analyzed and to identify potentially systemic issues in the system. Findings related to timing, scheduling, and fault tolerance and the benefits of the use of the AADL are examined. The report also highlights the benefits of working with architecture abstractions that are reflected in the AADL notation, in particular the separation of architecture design decisions from implementation decisions. Such a lightweight architecture analysis is typically followed by a full-scale AADL model of the system with required and actual timing, performance, and reliability figures, and its analysis to determine whether the requirements are met.

1 Introduction

The Society of Automotive Engineers Architecture Analysis and Design Language (AADL) has been developed for embedded real-time systems that have challenging resource (size, weight, power) constraints, requirements for real-time response, fault tolerance, and specialized input/output hardware, and that must be certified to high levels of assurance [Feiler 03]. Its development is based on experiences in using MetaH, an embedded systems architecture notation and tool set prototyped by Honeywell under DARPA funding [Feiler 00]. Intended fields of application are avionics systems, flight management systems, space applications, automotive applications such as engine and power train control systems, robotics applications, industrial process control equipment, and certain medical devices.

The language is used to describe the structure of an embedded real-time system as an assembly of software and hardware components. The language supports specification of functional component interfaces (such as incoming and outgoing data streams) and non-functional aspects of components (such as sampling rate, response time, degree of redundancy, fault characteristics, space and time partitioning to address fault containment, as well as safety and certification properties). The language describes the composition of and interaction between application components (such as how sensor data streams are processed by filters and controllers before being fed to actuators), and how these application components are assigned to processors in the execution platform.

The AADL is an emerging standard that was developed under the auspices of the International Society for Automotive Engineers (SAE) in their Avionics Systems Division (ASD) [SAE AADL 04]. It was balloted and accepted in May 2004. This standardization effort is led by Bruce Lewis, U.S. Army Munitions and Chemical Command (AMCOM); SAE AS2C Subcommittee Chair; Peter Feiler, Software Engineering Institute (SEI), editor and lead author; Steve Vestal, Honeywell, coauthor; Ed Colbert, University of Southern California (USC) and Absolute Software, author of the Unified Modeling Language profile for the AADL; and Joyce Tokar, Pyrrhus Software, author of the Programming Language Compliance Annex. Other participants in the standardization effort range from U.S. organizations such as Rockwell, Lockheed Martin, Boeing, Raytheon, Smith Industries, General Dynamics, National Institute of Standards Technology (NIST), U.S. Army and Navy to European partners such as Airbus Industries, European Space Agency, British Ministry of Defence (MOD), Axlog, Dassault Aviation, TNI-Valiosys, and European Aeronautic Defence and Space Company (EADS).

2 AADL Overview

The AADL is a modeling notation with both a textual and graphical representation. It provides modeling concepts to describe the runtime architecture of application systems in terms of concurrent tasks and their interactions as well as their mapping onto an execution platform. The AADL offers *threads* as schedulable units of concurrent execution, *processes* to represent virtual address spaces whose boundaries are enforced at runtime, and *systems* to support hierarchical organization of threads and processes. The AADL supports modeling of the execution platform in terms of *processors* that schedule and execute threads; *memory* that stores code and data; *devices* such as sensors, actuators, and cameras that interface with the external environment; and *buses* that interconnect processors, memory, and devices.

Threads can execute at given time intervals (*periodic*), triggered by events (*aperiodic*) and paced to limit the execution rate (*sporadic*), by remote subprogram calls (*server*), or as background tasks. These thread characteristics are defined as part of the thread declaration. Application components interact with other application components and devices exclusively through defined interfaces. A component interface consists of ports for unidirectional flow of data (data ports for unqueued state data, and event data ports for queued message data) and events (event ports) between threads and to and from devices; synchronous subprogram calls between threads, possibly located on different processors; and access to data that is concurrency controlled. Data port connections can be specified to perform mid-frame (*immediate*) communication within the same dispatch period, or phase delayed (*delayed*) communication for data to be available after the deadline of the originating thread. These semantics ensure deterministic transfer of data streams between periodic threads—an important characteristic for embedded control systems. Deterministic transfer means that a thread always receives data with the same time delay; if the receiving thread is over- or under-sampling the data stream, it always does so at a constant rate.

AADL components can have multiple modes. A mode represents a particular configuration of subcomponents and interconnections. Different modes may represent different operational configurations. Mode transitions specify event arrival conditions that cause mode switches to occur.

Application components have properties that specify timing requirements such as period, worst-case execution time, deadlines, space requirements, arrival rates, and characteristics of data and event streams. In addition, properties identify the following: the source code and data that implement the application component being modeled in the AADL; and constraints for binding threads to processors, and source code and data onto memory. The constraints can limit binding to specific processor or memory types, for example, to a processor with DSP support, as well as prevent collocation of application components to support fault tolerance.

The AADL can be used to model and analyze existing systems and to design and predictably integrate new systems. AADL models can be used for both analysis and construction throughout the life of a system. The AADL is extensible in that new properties and specification notations (e.g., constraint notation, reliability model), can be added (via language Annexes) to accommodate analyses not directly supported by the core AADL. Further details regarding the AADL can be obtained by referring to the AADL standard [SAE AADL 03].

The AADL also has a graphical notation to facilitate a visual presentation of the system hierarchy and communication topology. The graphical symbols used in this document are summarized in Figure 1. Some graphical symbols include an icon to indicate that its semantics differ from a similar graphical symbol in the Unified Modeling Language (UML). Some symbols are decorated with additional icons such as circles to represent component properties such as the period of a thread.

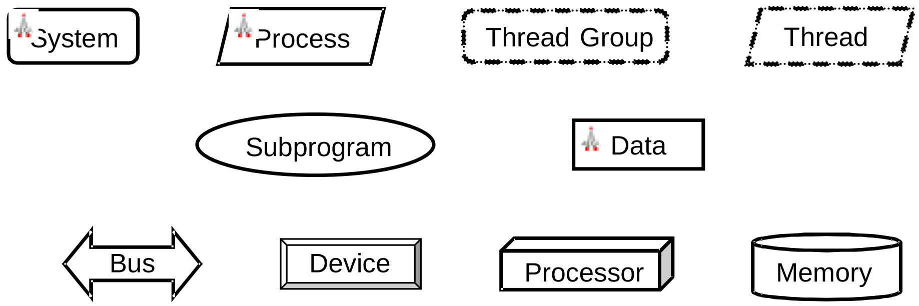


Figure 1: AADL Graphical Icons

The AADL can be used in two ways: lightweight analysis of architecture patterns identified in real systems to discover potentially systemic issues, and full-scale analysis of a complete system model with fully quantified system property values. In this technical note we focus on the use of the AADL as an effective tool for initial analysis of embedded systems for potential problem spots. We do so by focusing on different aspects of the embedded system architecture and identifying potentially unanticipated side effects:

- the migration from a statically scheduled system to a preemptively scheduled system to improve resource utilization and create a flexible architecture
- the impact of this change in task scheduling on task communication via shared variables
- the processing of system events by polling
- the scheduling of system partitions as virtual processors, management of end-to-end latency
- the modeling of redundancy in a fault-tolerant architecture

We will examine each of these issues in the next sections.

3 The Avionics System

The Carnegie Mellon® Software Engineering Institute (SEI) applied the AADL to analyze an existing avionics system design. An avionics system typically consists of a collection of hardware and software that controls the flight, navigation, radio communication, and in the case of military aircraft, the targeting and weapons systems. Early generations of digital avionics systems consisted of embedded controllers executing on specialized hardware. As general-purpose processors became faster, controllers were implemented with application software executing with a static timeline and shared variable architecture. Use of shared variables minimized the memory footprint and resulted in efficient communication between components within a controller. This approach led to an efficient implementation with deterministic execution behavior, but resulted in a software runtime architecture that was carefully crafted and difficult to change.

As avionics systems became more sophisticated, controller information had to be exchanged between avionics subsystems, and the systems have been continuously evolving. On the hardware side, high-speed bus architectures have been introduced but weapons systems and other devices still depend on the slower 1553 bus. This has led to a hybrid execution platform with some processors connected only to the high-speed bus, while others act as gateways to legacy hardware (see Figure 2 below).

On the software side, this led to the need for better resource utilization[1] of the available processors, and for a more flexible and distributed software runtime architecture. Preemptive fixed-priority scheduling and rate-monotonic scheduling analysis (RMA) have been introduced by commercial vendors [Klein 93]. This improves resource utilization and offers more flexibility in adding and re-allocating tasks across processors while using design time analysis to ensure that critical real-time deadlines are met. Modularization of the application and port-based communication has facilitated distribution of application components.

Embedded avionics software systems use *partitions* as virtual machines to achieve space and time partitioning [ARINC 653]. A partition provides runtime address space boundary enforcement; that is, code executing in one partition cannot inadvertently overwrite data or code in another partition. Such runtime enforcement of the address space is found in both Portable Operating Systems Interface (POSIX) processes and in the AADL process concept. A partition provides time partitioning by guaranteeing each partition a specified amount of real processor time. A faulty thread in one partition that exceeds its worst-case execution time

® Carnegie Mellon is registered in the U.S. Patent and Trademark Office.

[1] Better resource utilization means being able to execute a larger number of tasks while continuing to meet all deadlines.

cannot affect the execution of threads in other partitions. This is accomplished by scheduling the partitions on a static timeline. In other words, within a given time frame each partition sharing a processor will be given the processor for a specified amount of time. Each partition can then allocate this time to its tasks (called processes in Aeronautical Radio Incorporated [ARINC] 653 and threads in POSIX and AADL) according to its scheduling protocol. Tasks (a set of routines that execute within a partition) communicate across partitions through messages. This permits tasks to be relocated across processors for load-balancing purposes.

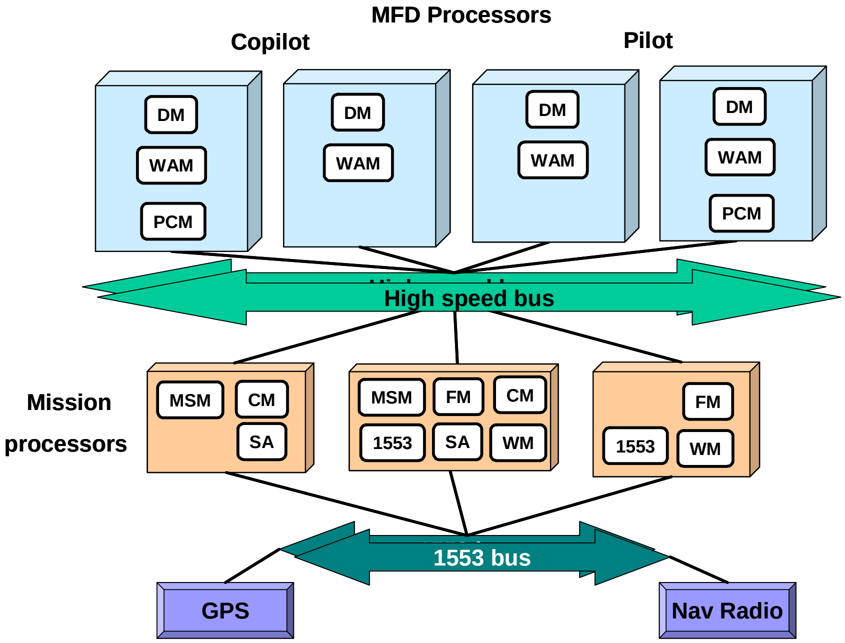


Figure 2: Typical Documentation of Avionics System Architecture

A typical diagram of such a software architecture mapped onto the hardware is shown in Figure 2. The figure shows two multifunction displays (MFDs), one for the pilot and one for the copilot. The MFDs provide menu-based access and data entry for different avionics subsystems and display warnings and alerts. Each MFD has a display processor that can host application functionality as well. Each application subsystem is implemented as a separate partition. The following application subsystems are shown as hosted by the MFD processors: MFD display manager (DM), warning and annunciation manager (WAM), page content and menu manager (PCM), and flight director (FD). The mission processors host the communication manager (CM), weapons manager (WM), situational awareness (SA), flight manager (FM), mission sensor manager (MSM), and the software interface to the MIL-STD-1553B communication bus (1553) and the devices hosted by it.

These types of 'architecture' drawings tend to raise more questions than they answer. For example, are the four DMs redundant or do they represent separate functionality? What are the delays (processing and communication) of certain signal paths? When a pilot selects a display to show engine-operating characteristics, how is the command processed and from where are the data values obtained? The answers to these types of questions are typically embedded in textual descriptions throughout the design document. We will explore these questions in the following analysis sections based on an AADL model of the avionics system.

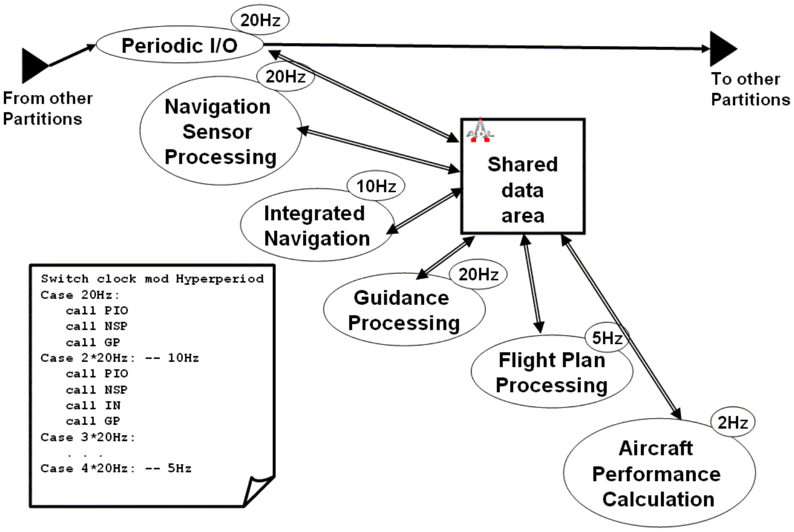
4 Preemptive Scheduling and Port Communication

In the following discussion, we will focus on a flight manager subsystem executing within one of the partitions. This subsystem consists of several components that process signal data in a certain order, with some components operating at 20Hz while other components operate at lower rates.

The focus of this section is the use of preemptive fixed-priority scheduling to achieve better resource utilization and a more flexible system design. In addition, port communication between threads within a partition is used to achieve more modifiable and configurable system designs while maintaining predictable and efficient execution and communication.

4.1 Shared Data and Fixed Execution Order

In many cases legacy implementations of embedded systems are simply augmented with a logical thread that performs communication with other partitions via send and receive routines, sending data from the common data area and placing the received data into the common data area.



```
Switch clock mod Hyperperiod
Case 20Hz:
   call PIO
   call NSP
   call GP
Case 2*20Hz: -- 10Hz
   call PIO
   call NSP
   call IN
   call GP
Case 3*20Hz:
   . . .
Case 4*20Hz: -- 5Hz
```

Figure 3: A Cyclic Executive Within a Partition

Communication with other partitions is shown in Figure 3 as a periodic input/output (I/O) thread. Communication between logical threads within the partition employs a common data area for two reasons: (1) the legacy code can be used unchanged; (2) this form of communication is highly efficient as no data is moved or copied between threads.

The execution order of the tasks is determined by the call order in the cyclic executive. Lower rate tasks are called at a lower frequency. The call order has been carefully crafted to achieve the desired flow order of signal data through the tasks. Figure 3 shows the call order visually by placement of the tasks from top left to bottom right.

Data flow information typically has to be inferred from the call order and a description of the data elements accessed by different components. This information is often located in different parts of the design document. Figure 4 shows each data element explicitly and indicates which tasks read from it or write to it. This makes data flows more apparent. For example, Guidance Processing passes data to Flight Plan Processing and also receives data from Flight Plan Processing. Given the call order, data is passed to Flight Plan Processing in one processing step (minor frame) and received in the next processing step.

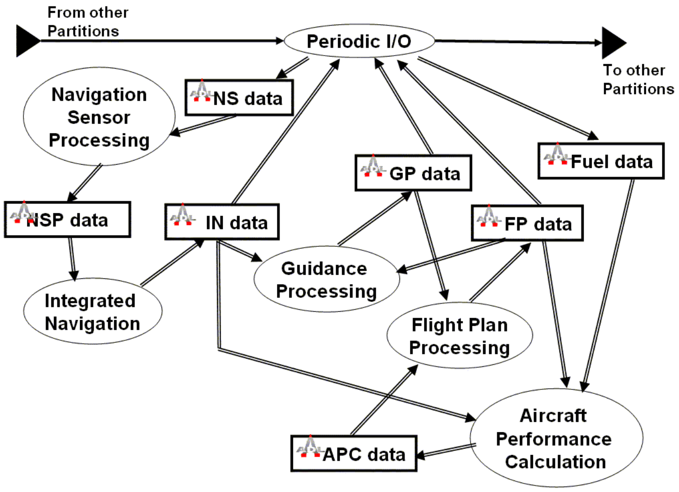


Figure 4: Data Flow in Form of Read and Write Access

In a cyclic executive, tasks are invoked in sequence and must complete execution before the end of the minor frame. The execution of one task cannot preempt the execution of other tasks. As a result, a task always sees data from another task either from the previous minor frame or from the current minor frame; that is, data communication is deterministic.

4.2 Use of Preemptive Fixed-Priority Scheduling

Preemptive fixed-priority scheduling is offered as a solution for improving resource utilization of processors and to increase the flexibility of evolving embedded systems while ensuring that deadlines are met. In particular, if used with RMA, a system design can be analyzed at design time to determine whether all deadlines will be met despite the fact that tasks can preempt each other.

A naïve way of introducing preemptive scheduling into this example is to turn each task into a separate thread (shown as a dashed parallelogram in Figure 5). To ensure the desired flow of data between components, priorities are assigned to the threads according to the desired execution order. The result is in priority inversion; that is, a lower rate thread has a higher priority than a higher rate thread. For example, the lower rate Integrated Navigation task is given a higher priority than the higher rate Guidance Processing task. This priority inversion does not occur if all threads can complete their execution in a minor frame; that is, they have a pre-period deadline corresponding to the highest rate thread. In that case deadline-monotonic analysis, a variant of RMA, ensures that deadlines will be met. A consequence of this assumption is that no thread is preempted and thread execution is the same as that of a static timeline. In other words, assigning priorities to enforce an execution order incurs the runtime overhead of preemptive scheduling without obtaining the benefits of improved resource utilization and flexibility.

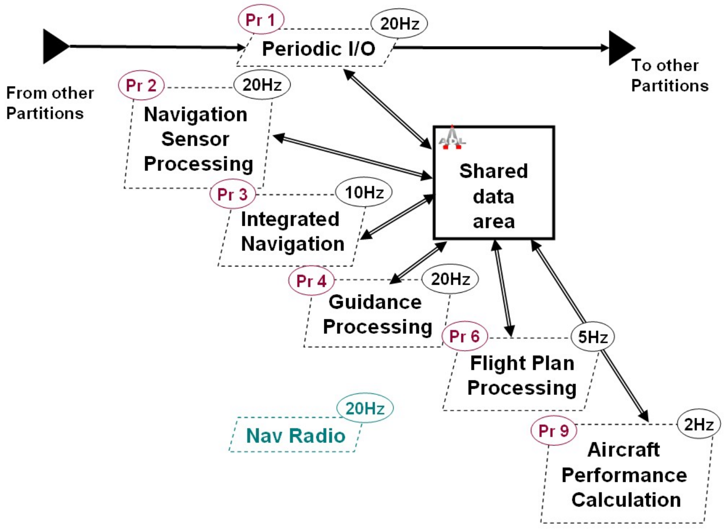


Figure 5: Preemptive Scheduling with Priority Assignment

A better way is to assign priorities according to task rates (rate-monotonic scheduling). In that case, tasks can potentially be preempted. For example, the Integrated Navigation task in the example may be preempted by a new dispatch of the Guidance Processing task. This can occur if, as a result of a code change, a component execution time increases.

However, preemption of a thread that communicates with other threads through a common data area introduces two potential problems: communication of consistent data and non-deterministic transfer of data. For example, the preempted thread may be in the middle of reading several data elements when it is preempted by a thread that updates those data elements. As a result, the preempted thread may have read the old value of one data element and the new value of another data element; that is, it may operate on inconsistent data. This problem can be addressed by ensuring that a thread reads or writes all data as an uninterruptible operation or by setting a mutex lock.

Non-deterministic data transfer occurs if a receiving thread sometimes receives old data values and sometimes receives new data values. This can occur if a lower rate thread sometimes is preempted by a receiver thread before it writes the new data values, and sometimes after it writes the new data values. This means that the receiving thread sometimes receives data within the same minor frame and sometimes it is phase delayed. In some cases, the recipient is insensitive to such non-determinism. In other cases the recipient compensates for the non-determinism; for example, to a controller it may look like noise in the sensor data stream. Finally, the non-determinism could result in undesirable consequences; for example, if the recipient displays a target position, the non-determinism could cause the displayed position to oscillate, blurring the display.

4.3 Port-Based Communication

Development of embedded systems applications, whose components interact through port communication, is becoming accepted practice. The AADL promotes port-based communication between all application threads, both within and across partitions. Furthermore, it distinguishes between queued message communication and unqueued state communication. Finally, the AADL distinguishes between immediate (mid-frame) and delayed (phase-delayed) communication of state data between periodic threads in a deterministic manner. Such communication semantics can also be found in real-time OS standards such as OSEK [OSEK 03]. In this section, we model communication within the flight manager partition through ports and discuss the issue of efficient communication implementations.

AADL *data ports* represent unqueued transfer of state data with the following semantics. From the perspective of application code, AADL ports are data variables. An *out data port* of one thread can be connected to an *in data port* of another thread. For periodic threads the connection can be declared to be delayed or immediate. In the case of a delayed connection, the value of a recipient's *in port* is set at dispatch time to be the most recent value made available from the sender's *out port*. This is the out port value at the most recent deadline of the out port thread. In other words, the received data value is phase delayed by the period of the receiving thread. In the case of an immediate connection, the transfer semantics are those of mid-frame communication. If both threads are dispatched at the same time, the thread with the out port executes first. At its completion, the data is transferred to the *in port*. Although dispatched at the same time, the execution of the receiving thread is delayed until the completion of the sending thread. Once a thread executes its *in port*, values do not change, even if the thread with the connected *out port* provides a new data value. This ensures deterministic data communication between periodic threads.

The AADL-based model of the flight manager is shown in Figure 6. All data communication is modeled by ports (black triangles) and connections; there is no need for shared data and coordinating concurrent access through locks. No task priorities have been specified by the modeler. They are determined according to the scheduling protocol; in the case of rate monotonic scheduling, according to the thread periods.

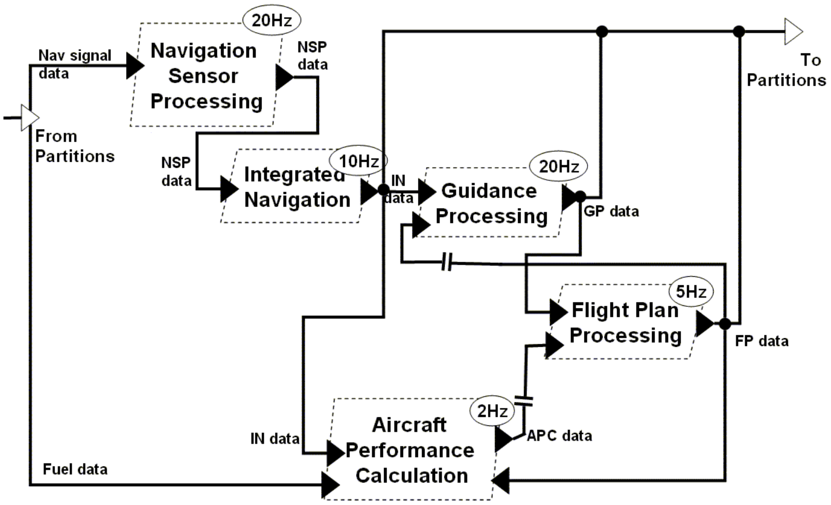


Figure 6: Port and Connection Based AADL Model of the Flight Manager

The model indicates which connections are *immediate* (solid line) and which are *delayed* (solid line with crossing double line). Cyclic sequences of immediate connections are not permitted since they cannot be achieved. Such cycles can be detected by an analysis tool. If the application developer documented an acceptable phase delay for a task (in a port property) the degree of actual phase delay can be calculated and compared against the acceptable value (see Section 7.1).

Note that the periodic I/O task is not represented explicitly in the model. The periodic I/O task achieves two objectives: (1) it groups several data items together and sends them as a composite data item (that is, the values of several output ports are sent together); (2) it always sends the data phase delayed at the start of the next period. In the AADL, these two concerns are modeled separately. Time-consistent data transfer of multiple *out data ports* is modeled by an aggregate data port (shown as hollow triangle), and phase delay as delayed connection. The application developer now has the choice of transferring the data immediately or delayed, by choosing the appropriate connection symbol.

The implementation of port-based data communication can be as efficient as the use of a common data area. An AADL-based tool can determine whether two communicating threads can preempt each other by examining the life span of the ports involved in a connection for each thread dispatch. If the life span of the two ports does not overlap, the same memory location can be used for both the out port and the in port. For example, the immediate connection semantics ensure that, for two connected threads with the same period, the thread with the out port executes first and the thread with the in port executes second; moreover, the first thread does not execute again until the second thread has completed.

A thread takes data from an *in data port*, performs some computation, and places the result in an *out data port*. Normally, the *in data port* and the *out data port* are represented by two separate source code variables. In the AADL, a thread can declare a port to be an *in-out port*, represented by a single source code variable. If that is the case, then a sequence of several threads connected by immediate connections can be optimized to a single source code variable.

In the case of the flight manager example, there is a sequence of immediate connections that includes several 20 Hz threads and the 10 Hz Guidance Navigation thread. The semantics of immediate connections implies that a lower rate thread followed by a higher rate thread in the communication sequence must complete before the deadline of the higher rate thread for it to meet its deadline. As we observed in the previous section, this leads to a cyclic executive execution pattern. We can increase resource utilization by not requiring immediate connections between the threads—in particular between a lower rate thread and a higher rate thread. In the context of the AADL-based model, this can be easily analyzed by turning immediate connections into delayed connections. On one hand, scheduling analysis can determine any gain in resource utilization. On the other hand, the developers of succeeding components in the sequence can examine whether their component can accommodate the additional phase delay. Changing an immediate connection to a delayed connection affects the life span of ports. A tool can determine whether an overlap in life span has been introduced that requires the use of separate port variables.

4.4 Observations on the Use of the AADL

The AADL separates runtime architecture design decisions from implementation decisions, and application component development from architecture design. At the same time, it precisely specifies temporal properties of both task execution and communication in such a way that application developers (control engineers) can develop their components against documented assumptions regarding sampling rates, phase delay of data, and processing rates. The semantics of AADL periodic thread execution and data port connections ensure deterministic and consistent data communication. At the same time, implementation of task dispatching and communication can be delegated to tools. Such tools can generate task dispatch code and communication code that correctly implement the intended temporal semantics. In addition, these tools can produce highly efficient implementations by taking advantage of information and analysis results from the AADL model.

Separation of architecture design from implementation concerns allows a software system engineer to investigate alternatives that improve the performance characteristics of an embedded system in cooperation with control engineers. One example is control engineers analyzing the sensitivity of their controllers to variations in phase delay, while software system engineers identify improvements in resource utilization. Another example is sensitivity analysis by control engineers to changes in sampling and execution rates, while system engineers investigate the impact of rate changes on schedulability and resource utilization.

5 To Poll or Not to Poll: Event Processing

Many real-time system designs rely on fixed periodic execution schemes to ensure determinism. However, most application domains are replete with aperiodic behaviors. The cockpit is full of switches that result in state changes. Furthermore, the MFD provides a programmable set of ‘soft’ switches and selectors. The sources of these events are pilot actions such as changing navigation radio channel settings or calling up a new status display by menu selection. A common technique to service these events is by periodic threads monitoring state data and taking action when the state changes. This polling technique is used for two reasons: this is how the service was provided in cyclic executive implementations, and polling maintains the periodicity of threads to be scheduled. Since events potentially can arrive at non-deterministic rates, event-based processing can potentially generate a processing load high enough to result in missed deadlines for periodic sensor data processing tasks.

In this section, we examine the polling technique for event processing and discuss event-processing alternatives that can be explicitly modeled in the AADL.

5.1 Event Polling

In the avionics system example, there are a number of tasks whose purpose is to identify events and take action when they occur. One example is polling for the pilot to change the channel selection of the navigation radio (NavRadio). This is done by a 20Hz task that observes the switch and dial state of the physical NavRadio interface. This state change is then translated into a request sent to the actual NavRadio device that is attached to the 1553 bus. The polling rate is chosen to be high enough that no state change, (e.g., a flip of the switch) is missed. A second example is the task responding to menu selections on the MFD. A DM task polls the MFD touch screen state at a rate of 20Hz. This rate was chosen to keep response latency to a minimum. When a menu selection is detected by the polling DM task, information is sent to the PCM. The PCM determines which command was selected and contacts the appropriate subsystem to provide the appropriate page content. Once the content is received, it is built into a page to be displayed by DM. At that point, the pilot can select the next menu item.

Polling maintains a deterministic execution pattern with a well-defined reaction latency. Polling eliminates system overload due to a surge of events. In addition, polling rates can be adjusted to accommodate minimum state change intervals and to ensure minimum reaction latency. Polling is also used when input hardware devices do not produce interrupts in the system.

Polling also raises issues. In the example system, the NavRadio thread executes in the flight manager partition. The NavRadio task was considered to be less important than some of the periodic signal data processing tasks; therefore it was assigned a lower priority (see Figure 5). This resulted in additional priority inversion.

Polling can also result in significant inefficiencies in resource utilization. For example, consider the 20Hz task monitoring menu actions. If we treat the task as a bona fide 20Hz periodic thread to determine schedulability of the system, we reserve processor time corresponding to its worst-case execution time, that is, the execution time it takes to process an actual event. However, the menu event can only occur at a maximum rate of 4Hz since processing a menu event involves five inter-partition communication steps. This means that only 20% of the processor resource allocated to this task is utilized. Similarly, we may have as many as 50 threads monitoring various switches and dials in the cockpit. However, a pilot cannot activate all switches simultaneously.

5.2 Event Processing by Sporadic Server

The AADL provides aperiodic and sporadic threads and event or event data port connections for modeling event-based processing. Aperiodic threads are dispatched by the arrival of events (e.g., lack a bounded minimum interval between subsequent instances). If the thread is active on event arrival, the event is queued and the thread is redispatched immediately upon completion of the previous dispatch execution. Sporadic thread dispatches are also triggered by event arrival, but the dispatch rate has a specified lower bound. This means that when events arrive in rapid succession, their processing is paced to a specified maximum rate and events are temporarily queued to meet the maximum dispatch rate requirement. This limits the potential resource requirements of event-driven threads.

Schedulability of periodic threads with co-existing event-driven threads can be determined in two ways: 1) treatment of aperiodic threads as periodic threads, and 2) use of a sporadic server scheduling scheme. The event-driven threads can be treated as periodic threads with a period that corresponds to the minimum interarrival time of their events. However, if the event arrival rate is stochastic with spikes in arrivals, the resulting schedule reserves resources for event processing as if event spike conditions are the norm. However, in many cases event processing has a bounded response time requirement that does not represent a hard deadline.

The sporadic server-scheduling scheme has been introduced and is supported by AADL to improve resource utilization for event-driven threads. A sporadic server handles aperiodic and sporadic thread execution [Klein 93]. With respect to co-existing periodic threads, the sporadic server is treated as another periodic thread and RMA ensures that all deadlines are met. Aperiodic and sporadic threads only execute when the processor is allocated to the sporadic server; thus, they do not affect the schedulability assurance of the periodic threads. The sporadic server has a scheduling policy for servicing these event-driven threads to minimize response time. A sporadic server can be integrated into the system scheduler or implemented at the application level. Resource utilization for event-driven threads can be further improved without affecting the schedulability of co-existing periodic threads by the use of a slack-stealing scheduling technique [Binns 97].

In the avionics example, we can specify the NavRadio thread to be a sporadic thread. We can bound its dispatch rate to the shortest time it takes to process a NavRadio command. For example, querying the current channel setting requires the NavRadio thread to send the request to the device on the 1553 bus and wait for a response. This command latency can be determined from a flow specification in the AADL model. Thus, a realistic rate of command events for NavRadio may be much less than 20Hz. Furthermore, if rate monotonic scheduling is used the thread priority is determined by the thread period; thus, a potential cause of priority inversion is eliminated.

A sporadic server can handle event-driven threads that are dispatched by different event streams. Earlier we made the observation that some event streams cannot occur at the same time. For example, the pilot can flip only a certain number of switches at any one time. Furthermore, the operational mode of the system may limit the number of subsystems that are active at any one time and that require event-driven interaction. Operational modes and active subsystem configurations are modeled by AADL modes. Based on mode information we can determine realistic processing load requirements for a sporadic server and scale its resource allocation accordingly.

5.3 Observations on the Use of the AADL

Polling has been used as an approach to handle event processing in a way that maintains the predictability of periodic thread execution. High polling rates increase responsiveness to events, but they reserve unused processor resource to the polling thread, and as a result reduce the number of threads that can be accommodated on a processor while maintaining schedulability.

The AADL provides explicit support for modeling event-driven systems through aperiodic and sporadic threads. It supports scheduling techniques that maintain the schedulability of co-existing periodic threads. Its modeling support for flow specification and operational modes permits a system engineer to determine realistic event rates based on operational context and considerably improve processor utilization.

6 Hidden Timing Side Effects of Partition Scheduling

Partitions are placed in a particular order on the static partition scheduling timeline of a processor. Partitions may have to be rearranged on the timeline or reassigned to other processors to accommodate new tasks and partitions and to balance the load across processors. Such rearrangement of partitions is a delicate undertaking and may have hidden side effects. This section focuses on the effects of such rearrangements on inter-partition communications within and across processors.

6.1 Inter-Partition Communication Within a Processor

Let us first examine the issue for inter-partition communication within a processor. We have a static timeline with partition A executing before partition B for the same time frame, followed by the execution of partition A in the next time frame, as shown in Figure 7. Partition A has two threads t_1 and t_2 that can be executed in either order. Partition B similarly has two tasks t_3 and t_4. If a thread t_1 in partition A sends data to a thread t_3 in partition B, the data is transferred mid-frame, that is, within the same time frame (T0-T1). If thread t_4 sends data to thread t_3, the data arrives at t_3 at the next time frame, that is, phase delayed (shown as an explicitly marked delayed connection). In other words, the partition order determines whether the flow of data occurs within the same timeframe. This is similar to the task scheduling scenario in Section 4, where the task order affects flow of data through shared variables.

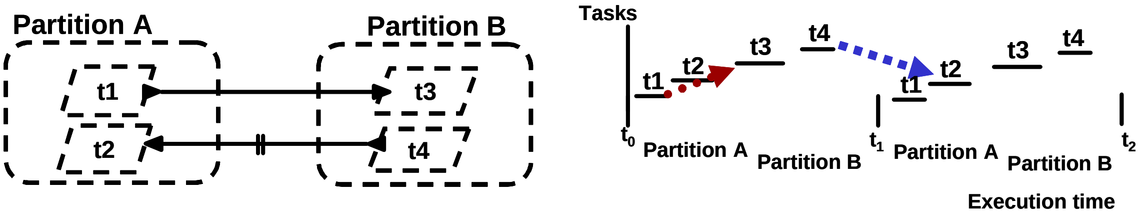


Figure 7: Partition Schedule and Communication

Modeling inter-partition communication in the AADL helps uncover a potentially undesirable side effect of rearranging the partition schedule. In the AADL, communication of state data is modeled by data ports and immediate or delayed connections. In other words, the desired timing characteristics of a connection are explicitly specified. These are the timing characteristics assumed by an application developer of a component executing in a task. These timing characteristics place a constraint on the possible partition orderings on the static partition execution timeline. Thus, a system engineer rearranging the partition timeline is made aware of such conflicts within the AADL description.

In the case of an immediate connection, the recipient partition must be placed after the sending partition. Note that there cannot be immediate connections from any thread in partition B to any thread in partition A. This can be easily detected through analysis of the AADL model. In the case of a delayed connection, the AADL semantics ensure that data will be transferred with a phase delay, independent of the execution ordering of the partitions. This means that for delayed connections, either the sender partition must be placed after the recipient partition, or the runtime system must double buffer the data to achieve the delay. Note that if a design is over-constrained, no partition order can satisfy the specified communication delay characteristics.

Both the application engineer and the system engineer can contribute to relaxing the constraints on partition ordering. The application engineer can design the system to use only delayed inter-partition communication. This is effectively the case in the system design of Figure 5, where the periodic I/O task performs all inter-partition communication at the beginning of partition execution. An application developer can also specify that a component is insensitive to (a certain variation in) phase delay, that is, that the connection could be either immediate or delayed, if the receiving component can handle variation in phase delay. The system engineer can provide an implementation of delayed inter-partition communication by transferring data just before a partition dispatch as part of the runtime system functionality, thus relieving the application developer from repeatedly implementing the periodic I/O task.

6.2 Inter-Partition Communication Across Processors

If we have a partitioned system that is distributed across multiple processors, the alignment of the static partition timelines on those processors determines whether communication is immediate or phase delayed. An AADL model of the application system will specify the desired communication timing characteristics, thereby placing constraints on the ordering of tasks on partitions across all processors. Techniques for relaxing the constraints on a partition apply on the assumption that the system is synchronous, that is, that the processors operate on a single global clock.

Processors in such a system may be connected via an aperiodic bus with data transferred immediately (with a well-defined maximum communication time), or via a periodic bus with data transferred at a rate determined by the bus itself. A periodic bus samples the data stream to be transferred and introduces a phase delay determined by the bus rate. This means that all connections that are bound to the bus must be delayed connections. In other words, only partitions with delayed data port connections can be placed on different processors that are connected physically by a periodic bus. This can be checked by analyzing the AADL model.

In a time-triggered architecture (TTA), the bus is periodic and drives the scheduling of tasks on different processors [TTA 03]. Thus, it acts as a global clock that manages any clock drift of individual processors. In that case, one can attempt to align the schedule of partitions across processors under AADL's immediate connection constraints. Again, the AADL model permits quick identification of over constraints due to immediate connections, for example, identification of immediate connections between two independent pairs of threads in two different partitions.

If a distributed system is asynchronous, that is, if each processor operates on a local clock, clock drift can occur. Two partitions with an immediate connection on different processors may have overlapping execution times and the ordering may change over time. In other words, their execution times relative to each other may vary over time, resulting in a varying sampling phase delay for the recipient. A periodic I/O task solution as discussed in Section 4 does not eliminate the non-determinism in phase delay due to clock drift. However, it does address the issue of time-consistent transfer of aggregate data, that is, the transfer of data as a single unit that is consistent with respect to the execution of multiple sending threads in a given partition. As mentioned earlier, the AADL provides an aggregate data port for this purpose.

6.3 Observations on the Use of the AADL

The ordering of partitions in a partition schedule can potentially affect the timing characteristics of connections. AADL models with immediate and delayed connections explicitly document the desired timing characteristics of data transfer. They act as constraints on the placement of partitions on their static timeline. This allows us to determine whether a feasible partition ordering exists. The constraints can be relaxed by the

- AADL runtime system's supporting delayed connections, independent of partition scheduling order
- application developer's investigating the
  - impact of a change of immediate connection requirements to delayed connection requirements
  - sensitivity of application components to variation in phase delay

The aggregate data port concept in the AADL contributes to addressing asynchronous distributed system issues by providing time-consistent data transfer.

7 End-to-End Latency

The avionics system has a number of flows, namely, signal streams that require periodic processing and aperiodic command processing flows such as changing the NavRadio channel. A critical requirement for these flows is to meet the maximum latency requirements. This requires end-to-end latency analysis. This end-to-end latency analysis can be based on

- deadline and worst-case execution time of individual steps in the flow executed by threads
- worst-case latency specified for the transfer of information from one step to the next

We can separately determine whether

- threads meet their deadline given their worst-case execution times for a given processor binding
- the bus can schedule the transfer of data for those connections that must communicate via the bus within their transfer latency limits

In this section we focus on end-to-end latency analysis on the assumption that the thread execution and data transfer performance properties have been validated.

Worst-case latency of a flow is effectively the cumulative latency along the path of a flow, that is, latency due to execution (competition for execution resources), communication (competition for the bus as resource), and sampling or pacing (delay due to dispatch delay and/or queuing delay). This can be based on the maximum execution latency and maximum communication latency figures. We can also consider average case end-to-end latency for those flows where it is acceptable.

7.1 End-to-End Latency Contributors

When determining end-to-end latency we distinguish between flows of unqueued data, such as signal streams communicated through data ports, and flows of queued data, such as commands sent as messages through event data ports.

Data streams through data ports can be processed by periodic threads or by event-driven threads. In the case of periodic threads, the data port connection between two succeeding processing steps may be immediate or delayed. If we have a sequence of periodic threads with immediate connections, the maximum latency of this sequence is determined by the deadline of the last thread in the sequence. Consider Figure 8a, next page. The top illustration shows the AADL representation of a set of periodic threads interacting by communicating a data variable from one to the other. For this case, assume that each component is mapped to its own thread of execution and that the execution time of any task is its worst-case execution time (WCET).

Given a set of periodic tasks τ_1, τ_2, $\tau_3\ldots$ τ_m,
with their request periods of T_1 , $T_{2,}$ $T_{3\ldots..}$ T_m ,
and associated deadlines of d_1, d_2, $d_3\ldots d_m$
and their execution-times being C_1, C_2, C_3, ….C_m
the latency (L) can be computed as $L = \sum_{n=1}^{m} C_n$.

Figure 8b shows the AADL representation for delayed communication, and underneath it is the graphical representation showing the effects of sampling delay. The end-to-end latency for a delayed set of connections can be expressed as

$$L_T = d_1 + \sum_{n=2}^{m} (s_n + d_n),$$

where *s* is the sampling time.

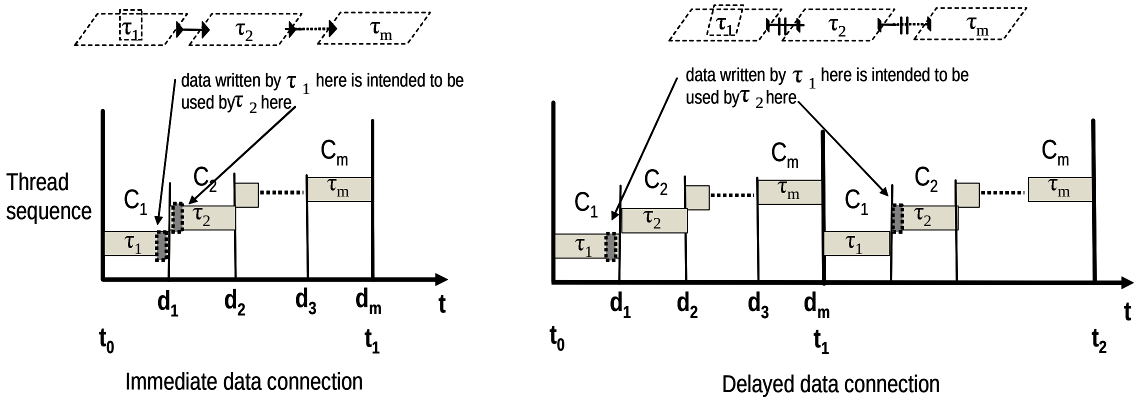


Figure 8: Cumulative Latency in Periodic Tasks for a) Immediate and b) Delayed Data Communication

If we have a delayed connection in a flow sequence through periodic threads, the recipient thread of the delayed connection samples that data stream at its period. It may extend the latency thus far determined to the next period; that is, it introduces a sampling latency based on its period. The latency in this case would be computed as follows:

$$L = \sum_{n=1}^{m} C_n + d_{n-1} - (d_{n-1} \bmod d_n)$$

where d_n and d_{n-1} are the deadlines of the sampling task and the task prior to the sampling task.

If we have event-driven threads in the sequence processing unqueued data, they contribute to the latency with their deadlines. In other words, if a periodic or event-driven thread passes data to its successor via a data port and triggers the execution of the successor with its completion, the successor thread has until its deadline to produce its output.

Thus far we have determined the logical end-to-end latency, that is, the end-to-end latency imposed by the application system architecture. This architecture can be modeled in the AADL with immediate and delayed connections, as well as flow specifications, to indicate the flow path from a source component to a destination component. When this architecture is bound to an execution platform, we may encounter additional contributors to the end-to-end latency. In Section 6.1 we identified partition ordering as a potential contributor, and in Section 6.2 we identified the periodic bus as a contributor.

If we have queued communication, we have to take waiting times in the queue into account. In the AADL, port queues are bounded by a specified size. This allows us to calculate the worst-case waiting time based on the processing deadline.

For queued communication it is more typical to determine average response times for flow paths. In that case the flow path can be mapped into a queuing model with statistical arrival rates and execution times, and averages can be determined through queuing analysis.

Recently, analysis approaches have emerged that limit (to an arbitrary precision) the probability of threads missing deadlines even if their inter-arrival times and execution times are stochastic. One such technique is known as real-time queuing theory [Lehoczky 96]. A small number of missed deadlines may be acceptable to successor steps in a flow path, since they already are accommodating incomplete data streams, for example, missing sensor readings due to intermittent problems. Such stream characteristics and the ability to accommodate them can be recorded in an AADL model through an extended set of properties.

7.2 Managing End-to-End Latency

When dealing with flows there are two major concerns: adjusting the end-to-end latency to meet requirements, and understanding the interaction between multiple flows, in particular at their merge points. In this section we examine both.

When actual end-to-end latency does not meet the requirements, a typical response is to ask application developers to make their code run more efficiently. However, this may be futile because certain latency contributors are inherent in the system or application architecture and are insensitive to a reduction in actual execution time by a thread. For example, consider output that is to be communicated over a periodic bus. Having a source thread execute faster to output a little earlier will not result in improvement unless the change crosses a period boundary of the bus sampling. Similarly, a periodic thread receiving data through a data port connection does not receive the data earlier if the sending thread is also periodic, since the data transfer semantics in that case are defined by the AADL to be deterministic (see also Section 4.3).

The representation of an application architecture in the AADL, with timing characteristics for both threads and connections and an explicit specification of flows, allows us to quickly identify the key contributors to end-to-end latency. In the previous sections, we have encouraged the consideration of delayed connections between threads to improve processor utilization and reduce constraints on partition scheduling order. These are decisions that can be revisited to reduce end-to-end latency. We may also eliminate sampling latencies if delayed connections can be turned into immediate connections. We can examine latency contributors due to the binding of the application system to the execution platform. For example, we can consider placing processing steps in a critical flow on the same processor. We can examine latency contributors due to allocation of application components into partitions. For example, we can consider collocating two sequential processing steps in the same partition.

A key issue with multiple flows is the interaction of their latency characteristics. If we have a periodic thread receiving data from an aperiodic thread, the actual completion time of the sending thread relative to the dispatch of the receiving periodic thread determines which value is accessible to the receiving thread. Variation in actual completion time may result in either the old or the new value being accessible; that is, data latency may non-deterministically vary by a period. This potential non-determinism can be identified through analysis and recorded as a property in the AADL model. Note that the semantics of immediate and delayed data port connections have been defined in the AADL such that neither immediate nor delayed data port communication between periodic threads introduces latency non-determinism.

Non-determinism in latency can result in potentially undesirable consequences. Section 4.2 discussed the example of an oscillating target position resulting in a blurred display due to the fact that the amount of phase delay (i.e., latency) varied. In general, whenever two data streams merge and one data stream has non-deterministic latency, there is a potential problem. In actual systems, the merge point is often a controller. In that case, any oscillation observed by the control engineer may be perceived as noise in the sensor data, for which the control engineer may compensate through adjustments in the controller.

7.3 Observations on the Use of the AADL

An AADL model specifies timing characteristics for both the execution of threads and the transfer of data between threads. The AADL supports the specification of end-to-end flows as well as flow specifications through individual components as part of their interface specification. As a result, the worst-case end-to-end latency of an end-to-end flow specified for a system can be determined in terms of the expected worst-case latency specified as part of the flow specification of each subsystem. In particular, this permits end-to-end latency analysis early in development to identify potential problem spots when subsystem implementations may not yet be completed. As the implementation of the system is refined, the latency analysis results can become less conservative to reflect the full implementation.

8 Redundancy in Application Architectures

Many embedded real-time systems have a requirement for high dependability. Dependability is the ability of a system to continue to produce the desired service to the user when the system is exposed to undesirable conditions [LaPrie 85]. One method to increase computer systems' dependability is through replication of hardware, software, or both. Critical hardware/software elements (or even complete systems) are replicated, to be brought into service when required. The AADL contains constructs that allow the developer to clearly represent and subsequently model the redundant artifacts at various levels of abstraction. In this section, we focus on the dependability aspects of a system and how general fault-tolerant approaches can be supported by the AADL.

8.1 Redundancy Described In Design Documents

In Figure 2, multiple instances of hardware and software are shown with little or no indication as to the intended functional redundancy. This results in speculation about the intended behavior of the system under fault conditions. Such information tends to be spread throughout the design document. For example, there are four MFD processors, four DMs, and four WAMs. Are they one operational unit with three spares, two operational units each with its own spare, or four fully functional operational units? What is the mechanism by which failures are detected? What is the mechanism by which failover is achieved? Does each replicated unit perform failover switching separately, or are groups of replicated tasks switched together? What data is necessary, if any, for state space preservation? What are the data sources that feed the redundant entities? Answers to these types of questions could not be ascertained from the architectural drawings. Reading through software design documentation uncovered some useful information, but not enough to completely model the system completely. It is in this setting that the AADL abstractions help guide us to a clear understanding of the fault-tolerant aspects of the system.

8.2 An Application Architecture Perspective of Redundancy

Analysis of this architecture from a dependability perspective begins with understanding what is being replicated. Having been provided with architecture drawings that intermix both hardware and software redundancy issues (Figure 2) one needs to sort out the intentions of the designers by asking exactly what functionality is to be redundant in the application system and in the execution platform, and what the events are that cause the redundant components to become active.

Through detailed review of some of the related system design documents and through discussion with system engineers, we were able to determine the following important aspects of the system (see Figure 9):

- In a normal operational mode the four MFDs and their processors are fully functional units providing services to the pilot and copilot independently. In a solo operational mode, they act as redundant pairs of systems in that either the pilot or the copilot can perform flight duties with his/her accessible MFD pair. Each of the MFDs has a separate DM with its own state. This is represented by four instances of DM.

- The PCM supplies each of the four DMs with page content independently (shown by separate port connections), while the WAM supplies all four DMs with the same set of alerts (fan-out from a single port).

- The WAM is a single functional unit with four replicated copies, and the PCM has two replicated copies (indicated by an appropriate redundancy property shown as 4X and 2X in the graphical view).

- The mission-oriented subsystems are dual redundant; their redundancy is managed in groups of three (shown by grouping them into a system marked with 2X). The FD is managed as a dual redundant unit by itself.

- All subsystems supply the WAM with alerts and the PCM with page content to be prepared for display. This is described by using the port group construct (shown as half circles), which reduces the amount of connection clutter at higher levels of the system architecture hierarchy.

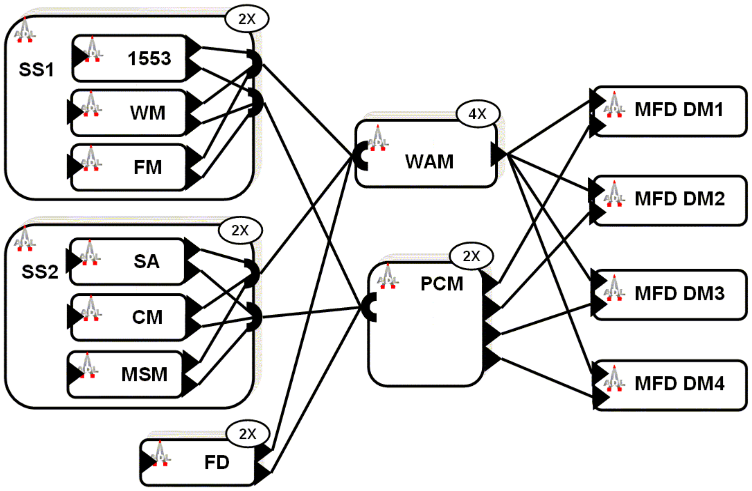


Figure 9: AADL Representation of Avionics System Redundancy

8.3 Modeling the Redundancy Protocol

Redundancy for fault tolerance involves the replication of hardware, software, or both. Where and what to replicate, the fault-detection mechanisms, and the control mechanisms to invoke the redundant entities are fundamental issues addressed within systems design. There are some common architectural approaches to redundancy. One approach is standby sparing (also referred to as dynamic redundancy, peer standby). In this fault-tolerant approach, one system is operating (e.g., in control) and the other units are spares, identically replicated and in some form of standby (e.g., hot, cold), ready to be switched into service when an unrecoverable error occurs. This is the design intention of the example that has been discussed thus far. Given that there are copies of software, this question follows: What is the operational scenario for failover? Detecting that a system in control has failed is a key problem with a number of known solutions (periodic tests, self-checks, watchdog timer, etc.). These techniques rely on events or data state changes that can be translated into an event to enable the switch to the spare. When the fault detection and switch over is carried out in the controlling (i.e., active) system and control is passed to a designated (passive) spare, this approach is termed *master-slave*. At a high level of abstraction, one is interested in the events that trigger execution of the spare.

We use the AADL mode concept to model alternative fault-tolerant system configurations. Figure 10 shows the replicated subsystem PCM as PCM.rep1 and PCM.rep2 contained in PCM, which takes on the role of SS1 (Figure 9). In master mode (shown on the left), PCM.rep1 is active, receives input, and provides output. PCM.rep2 (the slave copy) is not active and does not receive input nor produce output (shown in grey). In Slave mode (shown on the right), the opposite is the case.

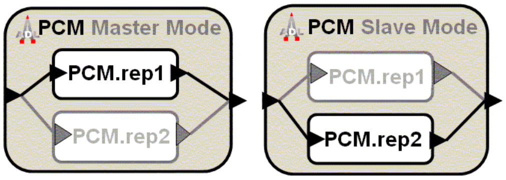


Figure 10: Standby Sparing, Active Master-Passive Slave

Figure 11 illustrates a hot-standby master-slave pattern of a stateful application component. In this case both copies of the component are supplied with input and both process the data. However, the output of only one copy is made available to the component output. The state of the component is modeled with the data component construct and is shown as exchanged between the components. This exchange can be specified to occur while operating in a mode, or on a mode transition. The figure also shows an Observer thread that receives the output from both copies and decides whether to operate in master or slave mode. The data is specified to be received by the observer thread at the next period. If a mode switch is necessary, it requests any necessary mode change by raising an appropriate event through the respective event out port (shown as a double arrow head). This event is routed to the appropriate mode transition in the mode state transition diagram. If the event arrives at an outgoing transition of the current mode, a mode switch is initiated.

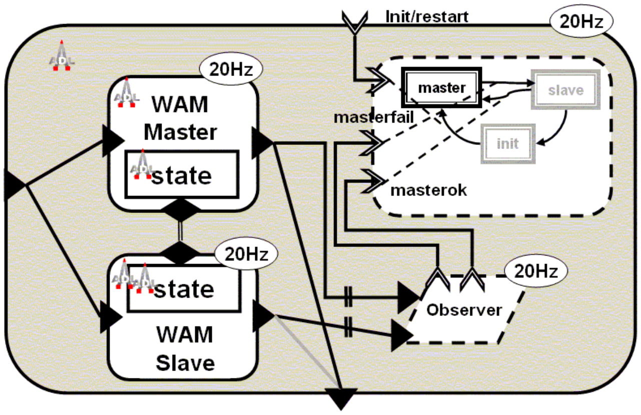


Figure 11: Hot Standby Master-Slave Mode Logic

Note that in Figure 9 we have abstracted the notions of application component redundancy into a set of properties. They indicate the degree of redundancy, the form of redundancy, and the desired redundancy protocol. Examples of the form of redundancy are replication as shown in the example, functional redundancy in the form of N-version programming [Avizienis 85], and analytic redundancy through functionally differing variants [Seto 98]. In this section we have shown how the chosen redundancy protocol can be modeled in the AADL.

8.4 A Runtime View of Fault-Tolerant Systems

Functional and analytic redundancy to address software faults can be achieved by executing the different copies on the same processor. However, addressing hardware faults by replication requires different copies to be located on different processors and memories. The AADL provides a set of properties that specify binding constraints of application components to execution platform components. These constraints can be in the form of limiting binding to certain processor or memory types and they can specify whether two components can be collocated.

In our sample system, the binding constraints specify that DM should be bound to a display processor for it to have local physical access to the display device. Similarly, the constraints specify that the 1553 subsystem must be located on a mission processor to have local physical access to the 1553 bus. To achieve effective fault tolerance we also specify that the redundant copies of the various replicated systems cannot be collocated. In the case of the two groups of mission-oriented subsystems, this constraint is specified for the aggregate system.

The AADL standard provides an error model extension that supports the description of a stochastic concurrent process reliability model through fault event rates. This model is transformed into a Markov chain for reliability analysis.

8.5 Observations on the Use of the AADL

The AADL allows the aggregation of application and execution platform components into a system hierarchy. Properties can be associated with components to specify the degree and form of desired redundancy. Redundancy protocols can be modeled in the AADL utilizing modes, mode transitions, and routing of events that reflect detected faults to appropriate mode transitions. Binding constraints address collocation restrictions of replicated components. Error models support stochastic modeling of fault occurrences for reliability analysis.

9 Summary

In this technical note, we have analyzed an existing avionics system to show use of the SAE AADL, an emerging international standard for modeling the system architecture of embedded real-time systems. The AADL focuses on modeling task and communication architectures by modeling application system architectures as threads, processes, and aggregates thereof, and by modeling their interactions as port connections, synchronous subprogram calls, or concurrency-controlled access to shared data. An application system architecture is then mapped onto an execution platform to support analysis of runtime system properties such as schedulability and reliability.

In the process of applying the AADL in the analysis of an existing avionics system, we were led to modeling the system so that implementation decisions were separated from architecture decisions. In particular, we were able to model the system interactions purely in the form of port communication, although the actual system is implemented with communication through shared variables. The use of the AADL abstractions allowed us to quickly identify potential issues with the shared variable communication solution within partitions.

The AADL model and its support for characterizing timing for both threads and connections allowed us to establish a framework for negotiating tradeoffs in resource demand between the application developer (typically, a control engineer) and the system engineer who is responsible for integrating the application components into an operational system. The characterization of connections as immediate and delayed also allowed us to identify issues with respect to partition ordering on the static partition scheduling timeline and permitted us to perform end-to-end latency analysis effectively.

Finally, the use of the AADL modeling capability allowed us to describe the redundancy aspects of the system architecture and to address fault tolerance concisely. By focusing on separation of concerns, we were able to describe the application system perspective, the realization of the chosen redundancy protocol, and the mapping onto the execution platform as three views.

References

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

[SAE AADL 03] Society of Automotive Engineers (SAE) Avionics Systems Division (ASD) AS-2C Subcommittee. *Avionics Architecture Description Language Standard*, Draft v0.99. Warrendale, PA: SAE, May 2004.

[SAE AADL 04] Society of Automotive Engineers. “The SAE Architecture Analysis & Design Language Standard Information Website.” http://www.aadl.info/ (2004).

[ARINC 653] Aeronautical Radio Inc. “Avionics Application Software Standard Interface,” ARINC Specification 653. Annapolis, MD: Airlines Electronic Engineering Committee, 1997.

[Avizienis 85] Avizienis, A. “The N-Version Approach to Fault Tolerant Software.” *IEEE Transactions on Software Engineering, SE-11,*12 (1985):1491-1501.

[Binns 97] Binns, Pam. “Incremental Rate Monotonic Scheduling for Improved Control System Performance.” *3rd IEEE Real-Time Technology and Applications Symposium.* Montreal Canada, June 9-11, 1997. New York, NY: IEEE Publishing, 1997.

[Feiler 00] Feiler, Peter; Lewis, Bruce; &Vestal, Steve. *Improving Predictability in Embedded Real-time Systems* (CMU/SEI-2000-SR-01, ADA387262). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. http://www.sei.cmu.edu/publications/documents /00.reports/00sr011.html

[Feiler 03] Feiler, Peter; Lewis, Bruce; & Vestal, Steve. “The SAE AADL Standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering.” *Workshop on Model-Driven Embedded Systems, Real-Time Application Systems (RTAS) Conference*. Washington, D.C., May 2003. See publications at http://www.aadl.info.

[Klein 93] Klein, Mark H.; Ralya, Thomas; Pollak, Bill; Obenza, Ray; & Gonzalez Harbour, Michael. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. New York, NY: Kluwer Academic Publishers, 1993.

[LaPrie 85] LaPrie, J.-C. "Dependable Computing and Fault Tolerance: Concepts and Terminology," 2-11. *Proceedings of the 15th International Symposium on Fault-Tolerant Computing (FTCS-15).* Ann Arbor, Michigan, 1985. New York, NY: IEEE Publishing, 1985.

[Lehoczky 96] Lehoczky, J.P. "Realtime Queueing Theory," *Proceedings of the 17th IEEE Realtime Systems Symposium*. Washington, D.C., Dec 4-6, 1996. New York, NY: IEEE Computer Society, 1996.

[OSEK 03] OSEK. *Open Systems And The Corresponding Interfaces For Automotive Electronics.* http://www.osek-vdx.org/ (2003).

[Seto 98] Seto, D.; Krogh, B.; Sha, L.; & Chutinan, A. "The Simplex Architecture for Safe On-Line Control System Upgrades," 3504-3508. *Proceedings of the 1998 American Control Conference*. Philadelphia, PA, June 24-26, 1998. Evanston, IL: American Automatic Control Council, 1998.

[TTA 03] *TTA: Time-Triggered Architecture*. http://www.tttech.com/ (commercial product) http://www.vmars.tuwien.ac.at/projects/tta/ (university research) (2003).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY
(Leave Blank)

2. REPORT DATE
June 2004

3. REPORT TYPE AND DATES COVERED
Final

4. TITLE AND SUBTITLE
Embedded System Architecture Analysis Using SAE AADL

5. FUNDING NUMBERS
F19628-00-C-0003

6. AUTHOR(S)
Peter H. Feiler, David P. Gluch, John J. Hudak, Bruce A. Lewis

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER
CMU/SEI-2004-TN-005

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

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

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

12B DISTRIBUTION CODE

13. ABSTRACT (MAXIMUM 200 WORDS)

The emerging Society of Automotive Engineers Architecture Analysis and Design Language (AADL) standard is an architecture modeling language for real-time, fault-tolerant, scalable, embedded, multiprocessor systems. It enables the development and predictable integration of highly evolvable systems as well as analysis of existing systems. It supports early and repeated analyses of a system’s architecture with respect to performance-critical properties through an extendable notation, a tool framework, and precisely defined semantics.

This report discusses the role and benefits of using the AADL in the process of analyzing an existing avionics system. The AADL is used to describe architecture patterns in the system being analyzed and to identify potentially systemic issues in the system. Findings related to timing, scheduling, and fault tolerance and the benefits of the use of the AADL are examined. The report also highlights the benefits of working with architecture abstractions that are reflected in the AADL notation, in particular the separation of architecture design decisions from implementation decisions. Such a lightweight architecture analysis is typically followed by a full-scale AADL model of the system with required and actual timing, performance, and reliability figures, and its analysis to determine whether the requirements are met.

14. SUBJECT TERMS
AADL standard, real-time, embedded systems, design-time analyses, schedulability analysis

15. NUMBER OF PAGES
41

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
Unclassified

18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT
Unclassified

20. LIMITATION OF ABSTRACT
UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18 298-102