

Pattern-Based Analysis of an Embedded Real-time System Architecture

Peter H. Feiler, Software Engineering Institute (SEI), phf@sei.cmu.edu¹
David P. Gluch, Embry-Riddle University, gluchd@erau.edu
John J. Hudak, Software Engineering Institute, jhudak@sei.cmu.edu
Bruce A. Lewis, US Army AMRDEC, bruce.lewis@sed.redstone.army.mil

Abstract

The emerging Society of Automotive Engineers (SAE) Architecture Analysis & 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. This paper discusses the role and benefits of using the AADL in the process of analyzing an existing avionics system. We use the AADL to describe architecture patterns in the system being analyzed and to identify potentially systemic issues in the system. We discuss some of the findings related to timing, scheduling, and fault tolerance and the benefits of the use of the AADL. Additionally we highlight 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 light-weight 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.

Introduction

The SAE Architecture Analysis & Design Language (AADL) [AS2C 04] 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. 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 AADL was developed under the auspices of the International Society for Automotive Engineers (SAE) in their Avionics Systems Division (ASD) and is in ballot as of March 2004 [AADL 04]. For more information on the AADL the reader is referred to www.aadl.info.

The AADL can be used as an embedded system engineering tool in two ways: analysis of architecture patterns identified in real systems to discover potentially systemic issues, and analysis of a full-scale system model with quantified system properties and generation of a model-specific runtime system [Feiler 03]. The SEI has applied the AADL to analyze an existing avionics system design as AADL patterns. The results of this work are summarized in this paper and described in more detail in [Feiler 04]. The cost-effectiveness of using MetaH, the precursor to AADL, for precise modeling, early analysis, and auto-generation of a system implementation is discussed in [Feiler 00].

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.

¹ Proceedings of Workshop on Architecture Description Languages (WADL04) August, 27 2004 - Toulouse France; part of IFIP World Computer Congress.

In this paper we focus on the use of the AADL as an effective tool for initial analysis of embedded systems for potential problem spots. We have analyzed the architecture of an avionics system that is being modernized. The analysis has focused on different aspects of the embedded system architecture and identify 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, scheduling of system partitions as virtual processors, management of end-to-end latency, and modeling of redundancy in a fault tolerant architecture. We will examine each of these issues in the next sections.

Preemptive Scheduling and Port Communication

In the following discussion, we will focus on a flight manager subsystem executing within one of the system 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 system is being migrated from a cyclic executive to the use of preemptive fixed priority scheduling to achieve better resource utilization and a more flexible system design. Preemptive fixed-priority scheduling is offered as a solution to 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 Rate-Monotonic Analysis (RMA) [Klein 93], 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.

Inter-partition communication port communication between threads is performed via message ports, while communication within threads is based on shared variables. The shared variable approach was retained to accommodate legacy components and to achieve highly efficient communication.

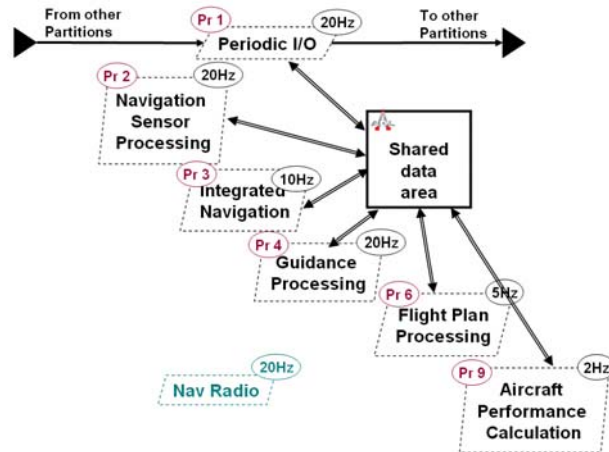


Figure 1: Preemptive Scheduling With Priority Assignment

A naïve way of introducing preemptive scheduling into this example is to turn each task into a separate thread. To ensure the desired flow of data between components, priorities are assigned to the threads according to the desired execution order. We have modeled this design using AADL (shown in Figure 1). The intended task execution order is from the top to the bottom. The task priority is indicated with Pr i , where a smaller i represents a higher priority.

Since AADL supports scheduling schemes based on RMA, it is natural to examine the resulting model from an RMA perspective. Thus, it is apparent that the manual priority assignment result in potential priority inversion, i.e., 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 potential priority inversion does not occur if all threads can complete their execution in a minor frame, i.e., they have a pre-period deadline corresponding to the highest rate thread. 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.

In summary, development of AADL models with an RMA-based fixed priority scheme provides properties for specifying the period, deadline, and worst-case execution time, but not for assigning priority. Thus, priority inversion cannot be introduced. If AADL is used to model existing implementations that do not use the RMA approach, then the red flag of priority inversion in the RMA framework can be provided as a consistency check in AADL models with explicit priority assignment.

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.

The AADL-based model of the flight manager is shown in Figure 2. 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.

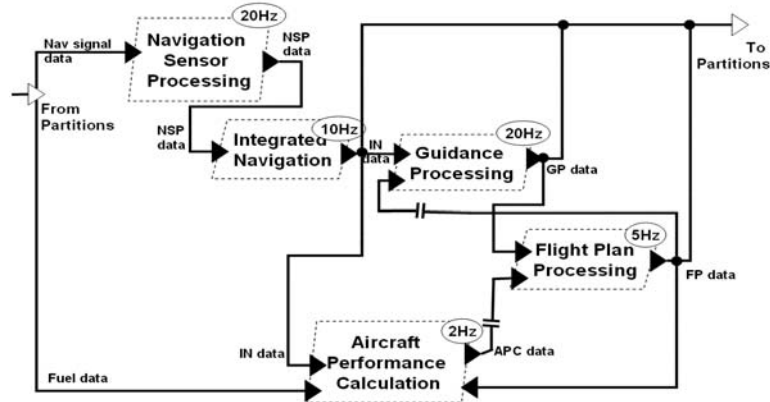


Figure 2: Port & Connection Based AADL Model of the Flight Manager

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.

Note that the periodic I/O task of Figure 1 is not represented explicitly in Figure 2. The periodic I/O task achieves two objectives: it groups several data items together and sends them as a composite data item, i.e., the values of several output ports are sent together; 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 as phase delay as a delayed connection. The application developer now has the choice of transferring the data immediately or delayed, by choosing the appropriate connection symbol.

The following observations can be made about the use of 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 assure 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 and communication code that correctly implements the intended temporal semantics. In addition, they 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.

Hidden Timing Side Effects of Partition Scheduling

Partitions provide time and space partitioning between software components. In doing so they ensure that malfunctioning components in one partition cannot affect the execution of components in other components. This concept is at the heart of the ARINC653 standard for avionics systems [ARINC653 97]. 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 rearrangements of partitions are 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.

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 3. Partition A has two threads t_1 and t_4 that can be executed in either order. Partition B similarly has two tasks t_2 and t_3 . If a thread t_1 in partition A sends data to a thread t_2 in partition B, the data is transferred mid-frame, i.e., within the same time frame. If thread t_3 sends data to thread t_4 , the data arrives at t_4 at the next time frame, i.e., phase delayed (shown as an explicitly marked delayed connection). In other words, the partition order affects the timing of communication.

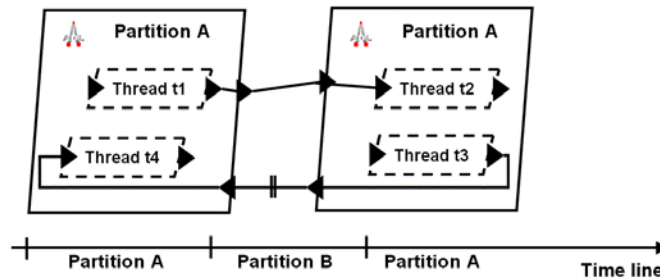


Figure 3: Partition Schedule & Communication

Modeling inter-partition communication in the AADL helps uncover a potentially undesirable side effect of rearranging the partition schedule. From an application perspective, flow between components is modeled as mid-frame (immediate) or phase-delayed (delayed) connections. 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 if there is an immediate connection from a thread in partition A to a thread in partition B (i.e., if partition A's execution must precede partition B's execution). This can be easily detected through analysis of the AADL model. 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 only use delayed inter-partition communication. This is effectively the case in the system design of Figure 1 by the periodic I/O task performing 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, i.e., 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. By doing so, the application architecture is insulated from partition ordering changes.

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 rely on the assumption that the system is synchronous, i.e., 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) [TTA 03] the bus is periodic and drives the scheduling of tasks on different processors. 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, e.g., identification of immediate connections between two independent pairs of threads in two different partitions.

If a distributed system is asynchronous, i.e., 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 shown in Figure 1, 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, i.e., 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.

In summary, the ordering of partitions in a partition schedule potentially can 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 supporting delayed connections, independent of partition scheduling order, and by the application developer investigating the impact of a change of immediate connection requirements to delayed connection requirements or the 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.

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 the flow of control information from sensors to actuators and changing the Navigation Radio channel. A critical requirement for these flows is to meet the maximum latency requirements. 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 and on the 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, and whether 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.

AADL supports the declaration of flows as flow specifications, i.e., as externally observable flows through components, as flow paths, i.e., the realization of the specified flows, and end-to-end flows, i.e., flow paths

with specific start and end points. Such flows are represented as sequences of connections and threads. From their timing characteristics as periodic threads with a given period, delayed and immediate connections, and whether connections are bound to periodic buses, we can derive the end-to-end latency. Worst-case latency of a flow is effectively the cumulative latency along the path of a flow, i.e., 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.

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. 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.

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, i.e., 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. For example non-deterministic variation in phase delay has the effect of an oscillating target position resulting in a blurred display. 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, which the control engineer may compensate for by adjustments in the controller.

In summary, 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 have been completed yet. As the implementation of the system gets refined the latency analysis results can become less conservative to reflect the full implementation.

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 02]. One method to increase computer systems' dependability is through redundancy of hardware, software or both. 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.

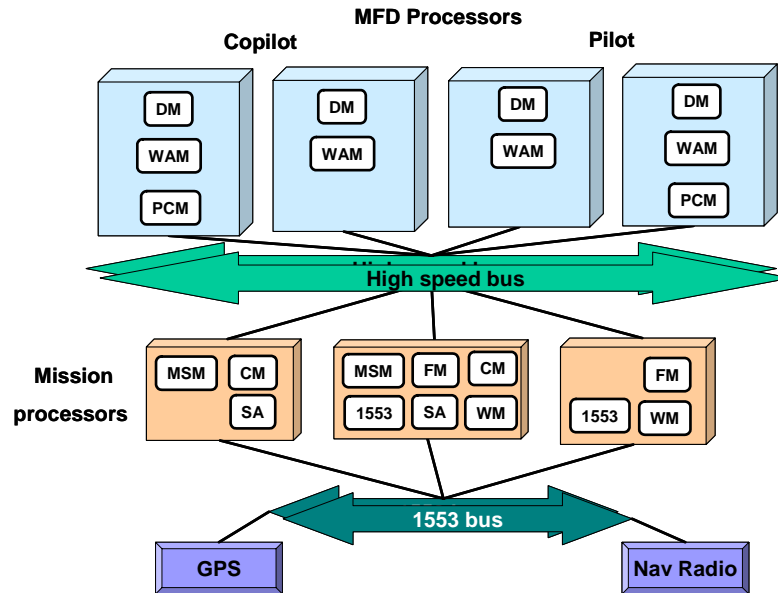


Figure 4: Typical Documentation of Avionics System Architecture

A typical diagram of such a software architecture mapped onto the hardware is shown in Figure 4. 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. It is in this setting that the AADL abstractions help guide us to a clear understanding of the fault tolerant aspects of the system.

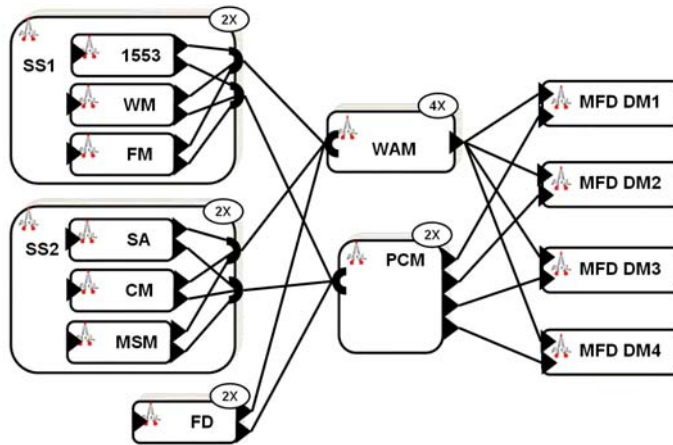


Figure 5: AADL Representation of Avionics System Redundancy

Analysis of this architecture from a dependability perspective begins with understanding what is being replicated. The intentions of the redundancy design can be expressed as a set of properties on the basic system architecture. In Figure 5 we are showing the above system as an AADL model. The logical grouping capability is used to clearly indicate which logical units are treated as redundant units. The degree of redundancy is indicated through a property shown visually in an oval decorator icon. Specific choices of redundancy mechanisms, such as master/slave, and the form of replication, such as n-version programming [Avi 85], are indicated through properties pre-declared as part of the AADL core language. Collocation constraints of components on processors and memory as well as connections over buses are similarly specified through properties.

Specific redundancy mechanisms can also be modeled in AADL as separate patterns. Figure 6 illustrates the master-slave pattern we found documented for several subsystems, each written with their own words and limited precision. This made it difficult to discern whether a single master-slave mechanism and protocol was used or whether different subsystems had variations. Questions that should be answerable from a design document include: Are both the master and slave active? What is the operational scenario for failover? Is state information exchanged between the redundant components? Who decides whether a component failed?

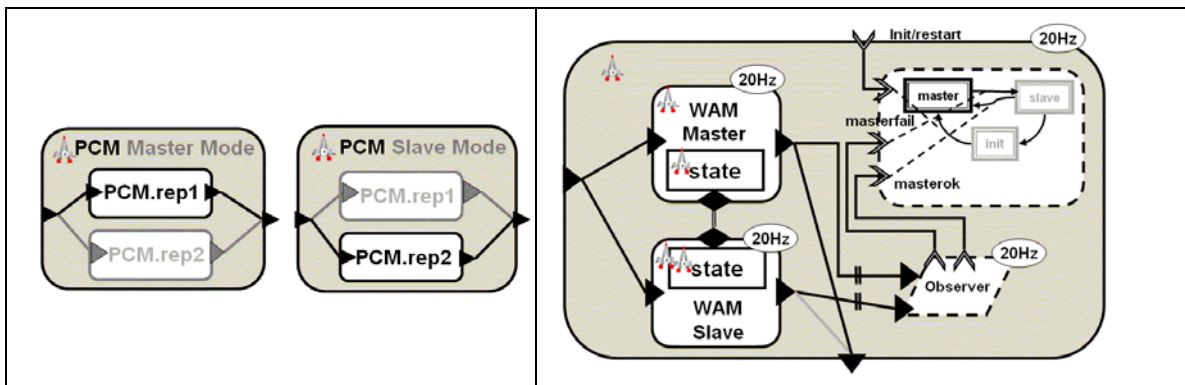


Figure 6: Hot Standby Master-Slave Mode Logic

We use the AADL mode concept to model alternative fault tolerant system configurations. Figure 6 shows the replicated subsystem PCM as PCM.rep1 and PCM.rep2 contained in PCM, which takes on the role of SS1 (Figure 5). The left hand side illustrates the different mode configurations of the master-slave pattern. In Master mode, PCM.rep1 is active, receives input, and provides output (shown in black). PCM.rep2 (the slave copy) is not active and does not receive input nor produce output (shown in grey). In Slave mode the opposite is the case. The right-hand side of Figure 6 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.

In summary, 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, 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.

Conclusion

In this paper, 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 April 2004.

- [AS2C 04] SAE AS-2c Subcommittee "Scope of the SAE AADL Standard". Excerpt from draft standard document. <http://www.aadl.info>
- [AADL 04] Society of Automotive Engineers (SAE) Avionics Systems Division (ASD) AS-2C Subcommittee. "Avionics Architecture Description Language Standard." Draft v0.98. Mar 2004.
- [Feiler 03] P. Feiler, B. Lewis, S. Vestal, "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, May 2003. See publications at <http://www.aadl.info>.
- [Feiler 00] P. Feiler, B. Lewis, S. Vestal, "Improving Predictability in Embedded Real-time Systems" Software Engineering Institute, Special Report, CMU/SEI-2000-SR-01.

- [Feiler 04] P. Feiler, D. Gluch, B. Lewis, J. Hudak, "Embedded System Architecture Analysis Using SAE AADL" Software Engineering Institute Technical Note CMU/SEI-2004-TN005, April 2004.
- [Klein 93] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, M. Gonzalez Harbour, "A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems," Kluwer Academic Publishers.
- [OSEK 03] Open systems and the corresponding interfaces for automotive electronics OSEK <<http://www.osek-idx.org>> (2003).
- [ARINC653 97] "Avionics Application Software Standard Interface", ARINC Specification 653, Airlines Electronic Engineering Committee, Aeronautical Radio Inc., 1997.
- [TTA 03] "TTA: Time-Triggered Architecture". <http://www.tttech.com/> and <http://www.vmars.tuwien.ac.at/projects/ta/>
- [LaPrie 02] J.C. LaPrie (editor), "Dependability: Basic Concepts and Terminology", Springer Verlag (publisher), November 2002, ISBN: 037822968.
- [Avi 85] A. Avizienis, "The N-Version Approach to Fault Tolerant Software." IEEE Transactions on Software Engineering, SE-11(12):1491-1501.