Aerospace Resource Document

REQUIREMENTS FOR THE AVIONICS ARCHITECTURE DESCRIPTION LANGUAGE (AADL)

Feb 15, 2001

REQUIREMENTS FOR THE AVIONICS ARCHITECTURE DESCRIPTION LANGUAGE
ARD 5296

Requirements for the Avionics Architecture Description Language (AADL)

This document was prepared for members of the AS-5C (Architecture) subcommittee of the Embedded Computing Systems committee, Aerospace Avionics Systems Division, SAE. These requirements are to be used by the subcommittee for the development of the AADL language standard. The AADL is designed for the specification of real-time, fault-tolerant, securely partitioned, dynamically reconfigurable multi-processor system architectures. It allows system architects to specify application software and execution platform architectures for computer control systems. Developers can use the AADL to specify how code modules written in traditional languages (e.g., Ada, C, Java) are combined to form an application, to specify the structure of a particular target platform, and to specify how the application software is allocated to target execution platform.

If more information is desired, please contact Bruce Lewis, US Army Aviation and Missile Command, AMSAM-RD-BA-AT, Hackberry Road, Bld. 6260; Redstone, Arsenal; Al. 35898 (e-mail: lewis@sed.redstone.army.mil).

The following were major participants and voting members in the development of this document: Robert Allen, Chandra Bhongir, Chris Dabrowski, Peter Feiler, Jeffry Fortin, Jim James, John Kraus, Jonathan Krueger, Bruce Lewis, Mike Pakucko, Diane Schleicher, Steve Vestal, Geoff Wilcock.

REQUIREMENTS FOR THE AVIONICS ARCHITECTURE DESCRIPTION LANGUAGE
ARD 5296

TABLE OF CONTENTS

1 SCOPE

This document lists requirements for the Avionics Architecture Description Language (AADL). The requirements are divided into two categories: core and annex. Core requirements shall not be compromised if the result is to be called the AADL. Annex requirements describe examples of extensions to the core AADL.

1.1 Purpose

The purpose of the Requirements Document is to constrain and direct the development of the AADL standard. The SAE Generic Open Architecture (GOA, see reference 4) model provides a framework for categorizing interfaces in an application architecture. We present a mapping to GOA to help those familiar with GOA to understand how the AADL fits with architectural concepts expressed in GOA. From the perspective of GOA, the AADL describes the interaction between time- and event-driven Application Software components. It couples this with a description of the computing and communication resources available on a target platform. Based on these descriptions, it is possible to completely generate and configure a set of Extended Operating System (XOS) Services and Resource Access Services tailored to the needs of the Application Software. The following diagrams (Figures 1, 2, and 3) relate AADL, as currently exemplified by the avionics architectural description language MetaH (ref. 6), to the GOA architecture.

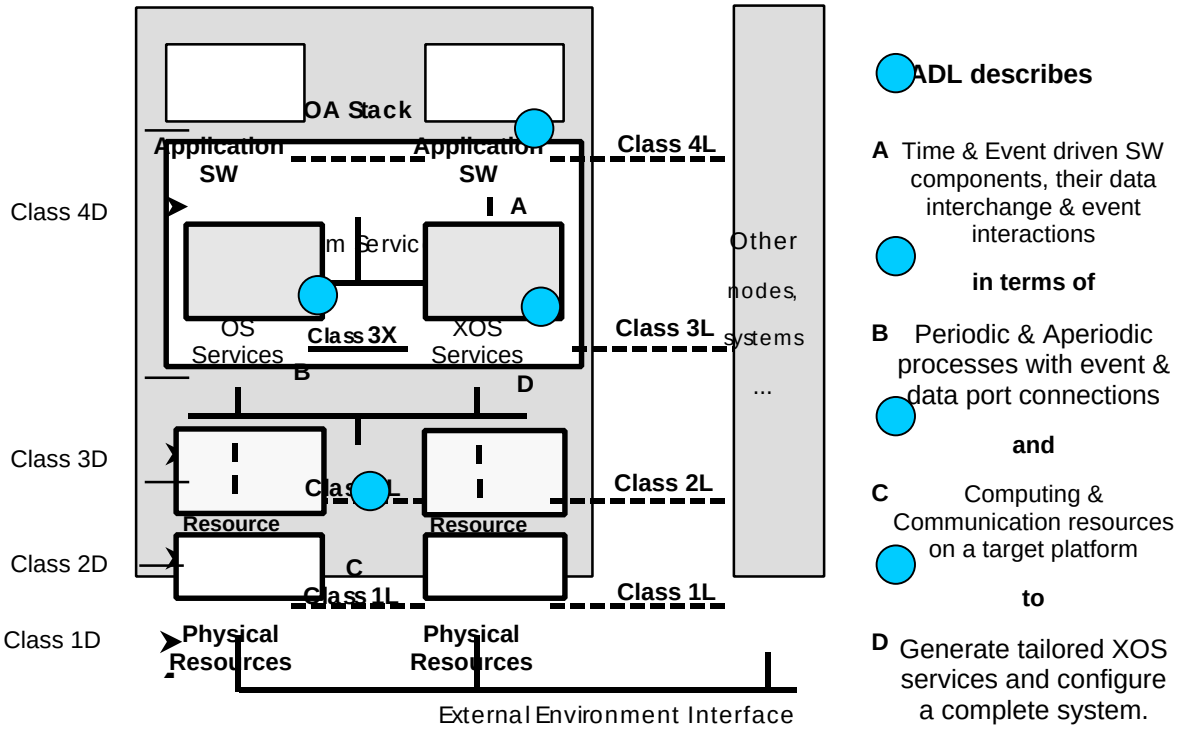

Figure 1. Relationship between AADL and GOA

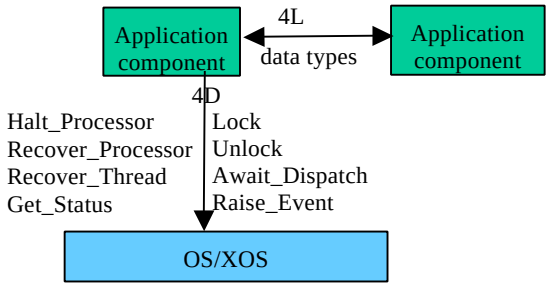

Figure 2. Application Component Source Code View

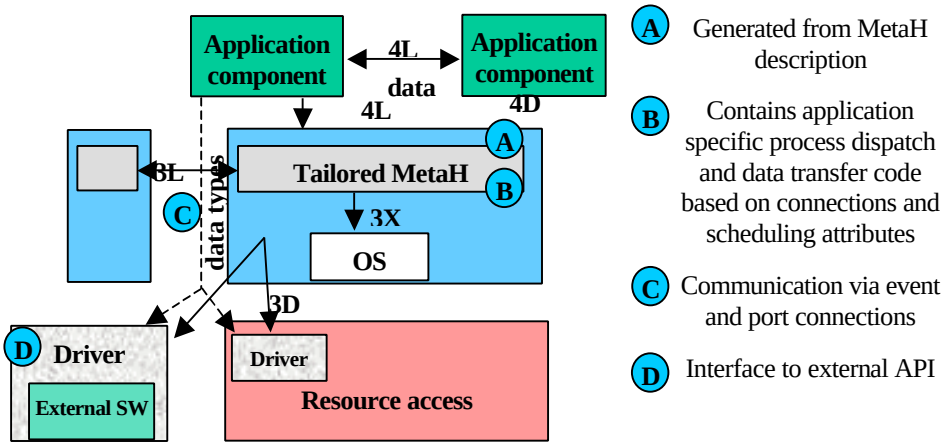

Figure 3. Notional Generated Runtime Architecture

1.2 Field of Application

The AADL is designed for the specification of real-time, fault-tolerant, dynamically reconfigurable, securely partitioned, multi-processor avionics system architectures. It allows system architects to specify application software and execution platform architectures for embedded avionics systems. Developers use AADL to specify how code modules written in traditional languages (e.g., Ada, C, C++, Java) are combined to form an application, to specify the structure of a particular target execution platform, and to specify how the application software is allocated to platform execution elements (e.g., processors).

The AADL is used to describe interfaces and other properties of code modules (but not their implementation details), and to specify how code modules are combined into higher-level objects such as processes and modes of operation. AADL also allows multi-processing platform architectures to be specified using descriptions of basic hardware objects such as processors, memories and inter-processor communication channels.

1.3 Benefits

Embedded systems software integration is in desperate need of improved architectural methodologies and tools. Avionics systems are being required to provide more functionality, such as on-board maintenance and diagnostics, multimedia communications, situation awareness, mission simulation and training, higher levels of cockpit automation, greater autonomy and less dependence on central traffic management, etc. Software applications previously deployed at arm's length on

federated boxes are moving onto "stock" multi-processing hardware resulting in hardware sharing among software applications. Engineers from a variety of specialties are thrust together to wrangle out architectural decisions with little means of assessing their impact.

An ADL helps by separating architectural issues from algorithmic issues. It describes the system as a set of interconnected components subject to an overall execution model. An ADL aimed at a specific domain (such as avionics) can give designers a way to check their system against particular properties required by their application. An AADL specification can be translated into executable "glue" code implementing the specified connections and execution model tying the components together.

The requirements for AADL were based on MetaH which was originally developed to meet the strict requirements of flight control and avionics, including hard real-time, safety, security, fault tolerance and multi-processing.[1]

AADL minimizes the impact of evolving execution platform and application software by explicitly representing key application/platform interfaces in a unified way across platforms, and by providing a basis for automatically customizing the XOS services to the application-specific architecture. Examples of architectural change accommodated by AADL are: process to processor binding, process and processor scaling, communication, processor type, bus type, fault recovery strategy, process rates, event communication, I/O ports, I/O rates, safety dependencies, process isolation and level of control, etc. AADL greatly simplifies the effort to get on the embedded target platform and to evolve the application and target. AADL reduces the verification and validation (V&V) required when components change by limiting change impact, and significantly reducing the cost of updating "glue" code.

2 REFERENCES

2.1 Applicable Documents:

1. RTCA DO-178B—Software Considerations in Airborne Systems and Equipment Certification, December 1, 1992, RTCA Inc., Washington, D.C.

2. IEEE Std 729-1983—Glossary of Software Engineering Terminology, 1983

3. IEEE Std 1471, Recommended Practice for Architectural Description, Draft Version 2.0: 5 May 1998 (Unapproved)

4. GOA—Generic Open Architecture Framework (AS 4893)

5. MetaH User Guide Version 1.29, January 1999, Honeywell, Inc. http://www.htc.honeywell.com/metah

6. Krueger, J., Vestal, S. and Lewis, B. (1998) "Fitting the Pieces Together: System/Software Analysis and Code Integration Using MetaH" in *DASC 98 Proceedings,* IEEE

2.2 Definitions

Architecture—The highest-level concept of a system in its environment (IEEE 1471, ref. 3)

Architectural Description—A product which documents an architecture (IEEE 1471, ref. 3)

Component—An element of an architecture that has one interface and one or more implementations. In AADL, a component may represent a portion of the application (software) or the execution platform (software and hardware).

[1] As other systems (e.g., robotics, automotive) increase their span of control, they share these requirements.

Configuration—The arrangement of a computer system or network as defined by the nature, number, and the chief characteristics of its functional units. More specifically, the term configuration may refer to a hardware configuration or a software configuration. (ISO/IEEE, ref. 2)

Domain-Specific—Specialized for a particular engineering discipline, industry, product family or other focus of concern.

Development Environment—A set of tools and processes that support the development of a product.

Embedded Computer System—A computer system that is part of a larger system and performs some of the requirements of that system; for example, a computer system used in an aircraft or rapid transit system. (IEEE Std. 61 0.12-1990)

Execution Platform—The hardware and software required to execute the application.

Interface—The means that two or more components (hardware or software) use to communicate with each other. This includes all the details of the communication protocol, including direction of data flow, structure, type and size of data, synchronization, format, variable names, etc.

Mode—The state of an application in terms of its active components and the connections between them.

Thread—A path (sequence of instructions) taken by a program during execution. It may proceed concurrently with other threads.

System—A collection of components organized to accomplish a specific function or set of functions (IEEE 610.12)

2.3 Acronyms and Abbreviations

AADL—Avionics Architecture Description Language

API—Application Programming Interface

GOA—Generic Open Architecture

OS—Operating System

XOS—Extended Operating System

3 OVERVIEW

An Architecture Description Language (ADL) describes the architectural characteristics of a system in terms of components, interconnections and system control. Domain specific ADLs can provide significant leverage to develop architectures with the correct properties for a domain of application, since the ADL description provides a basis for rigorous architectural analysis. An ADL can also provide the basis for a toolset that automatically implements the architecture specified.

The AADL will employ this paradigm for the development of embedded, high-assurance systems with hard real-time requirements. It will support development and evolution of systems by architectural specification. AADL will provide a highly automated, formal, architecture level language approach to building and evolving application software. It will support an architecture-centric process from the architectural specification to the implementation of the application software on the execution platform. AADL will permit rapid evolution and integration of components. It will provide a common architectural platform upon which domain specific code generation tools can build, such as display code generators, control code generators, signal processing generators, etc.

An important AADL goal is to support integrated modular avionics systems. Avionics systems are moving away from federated systems in which each function has a dedicated processor towards integrated systems in which multiple functions are hosted on a multi-processor. This significantly reduces logistics and maintenance costs, reduces size/weight/power, and makes it easier to upgrade functionality through software changes alone.

Integrated avionics applications are functionally heterogeneous. Most of the workload consists of multiple and diverse functions (e.g., autopilot, navigation, databases, diagnostics). However, some functions will increasingly be implemented using multiple, concurrent algorithms (e.g., guidance and route planning, image processing and display generation). Most on-board integrated modular avionics systems will be of modest size relative to massively parallel computers (e.g., tens rather than thousands of processors). Any parallelism will be relatively course-grained (e.g., a few parallel tasks that execute for several milliseconds each, rather than thousands of parallel operations of a few microseconds each). Further, many avionics systems have stringent size/weight/cost constraints.

Different functions in avionics systems are typically classified to be of different levels of criticality, e.g. safety-critical, mission-critical, non-critical. The higher the criticality level is, the more extensive and expensive the certification process is. Rather than certify all software to the highest level or provide separate processors for each criticality level, each function can be certified to the level required if the system guarantees that no defect in a less-critical function can possibly disrupt the operation of a more-critical function. The executive must not only provide memory protection (space partitioning) but also protection against disruption of real-time scheduling (time partitioning).

As mentioned above, a sufficiently detailed AADL description of a system will support automatic production of “glue” code composing the various software source modules to form the overall application. This glue code resembles an application-specific executive or supervisor, with code to dispatch processes and pass messages, synchronize access to shared resource, vector events, perform mode changes, etc. The glue code itself is expressed in terms of an underlying run-time or operating system. To support all of the aspects of AADL, the runtime/OS must provide memory protection, must be able to limit the capabilities of application components to make certain service requests, and must insure that the scheduling of high-criticality processes cannot be disrupted by the behavior of a lower-criticality process.

Figure four shows the role the AADL language will have relative to potential toolset functions and target systems. Graphical and/or textual editors can be used to create a model expressed in AADL. The model references and describes the architectural aspects of source components. Source components may be provided by various means, including hand-written, autocoded (e.g., generation from control block diagrams) and re-engineered. A partial or complete AADL model can be subjected to any number of analyses to yield metrics on key characteristics of the system. The model can also

be used to configure and produce an executable load-image whose resource allocations, processing rates, communication patterns, etc. are determined directly from attributes in the model.

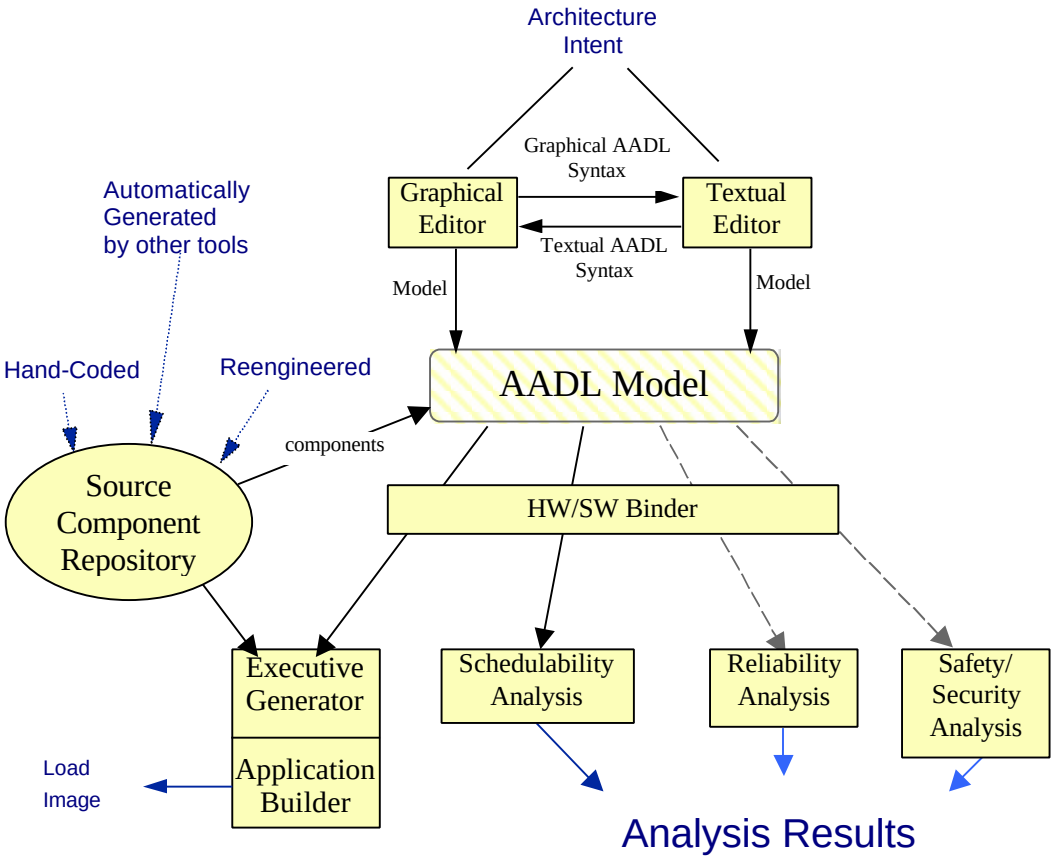

Figure 4. Notional Toolset

4 ESSENTIAL REQUIREMENTS

The requirements in this section are divided into 5 categories:

- System Architecture—The chief descriptive capabilities of AADL, focused on topology.
- Support for Real-Time Applications—Requirements about the role of time in AADL descriptions
- Embedding—Constraints on AADL to support the performance and footprint challenges of embedded systems.
- Life Cycle Support (of an application)—The role of AADL in system/software development processes
- Extensibility—How users and tool vendors can adapt AADL to special situations.

4.1 System Architecture

4.1.1 The AADL shall provide a means of defining application and execution platform components and specifying component characteristics and component interfaces. These components may be either hardware or software components.

Discussion: The aim of the component definitions is to provide a means of combining the components to make a system whose architecture can be examined, analyzed and produced. It is sufficient to describe each component as a black box with all algorithmic details hidden, but with all of its interface characteristics exposed. An interface characteristic is anything about a component that may affect the execution of other components, such as timing, data input/output, binding to execution elements, etc.

4.1.2 The AADL shall provide a means of specifying component interactions, including data dependencies and control dependencies.

Discussion: There are a variety of interaction topologies that must be considered, including 1-1, 1-many, many-to-1 and many-to-many. Each one of these has significant implications on the potential analyses and execution models. Dimensions of concern for data interaction include direction, data type, data size and synchronization. Dimensions of concern for control interaction include direction, event type, mode switching and processor allocation.

4.1.3 The semantics of the AADL shall be formal, supporting automated analysis and code generation.

Discussion: While intimated heretofore in this document, this requirement makes it official. Formality in this sense means mathematically analyzable and or executable.

4.1.4 The AADL shall permit analysis and code generation without the specification of component implementation details, e.g., algorithms, circuits.

Discussion: There are ample means available to designers for describing the functional characteristics of system components, which can change rapidly. AADL is aimed at describing architectural characteristics, which changes less dramatically over the lifetime of a system, while permitting rapid evolution of components.

4.1.5 The AADL shall support redundancy management approaches, including consensus protocols and voting algorithms.

Discussion: Not all avionics systems require redundancy, and for those that do, different mission profiles motivate different approaches. Nevertheless, fault-tolerant applications invariably include code that implements consensus protocols or voting algorithms.

4.1.6 The AADL shall support hierarchical decomposition to define the structure of a system, as well as a method for managing complexity.

4.1.7 The AADL shall support the inheritance of characteristics from one component type to another component type. The inheritance features of AADL shall be programming language independent.

Discussion: Inheritance is an increasingly popular feature of specification languages. It permits the definition of abstraction layers that strengthen reusability and economy of description. The primary consideration for AADL standardization is the liberty given to the child to override inherited characteristics, such as attributes, connections and interface components.

4.1.8 The AADL shall provide a means of specifying multiple component implementations.

Discussion: For any single system, each component typically needs only a single implementation. Over a product family however, components may require alternative implementations. Interconnections may be different, timing properties may be different, etc.

4.1.9 The AADL shall support the explicit definition of multiple modes of operation and the transition between modes.

Discussion: Modes of operation often correspond to different portions of an avionics system's mission (e.g., taxi, takeoff, cruise). For example, a mode could define which processes are active and, by implication, which data and event connections are relevant. (E.g., training mode versus tactical mode, taxi vs. takeoff.) Modes can also be hierarchical (modes can have submodes). It is conceivable that parts of a system running concurrently may have independent modal structures (parallel modes).

4.1.10 The AADL shall include parameters for modes, provided the parameters preserve the ability to analyze the specification.

Discussion: Forcing one to explicitly describe each potential configuration of execution threads and communication patterns (as in 4.1.9) is very limiting for some applications that would prefer to view modes as tailorable templates. Users should be able to concisely specify patterns of modes without specifying each system mode or configuration in its entirety. However, there is a tension here between flexibility and analyzability. Permitting too many degrees of freedom in mode specifications may render their analysis impossible or useless.

4.1.11 The AADL shall use implementation-neutral syntax to describe components.

Discussion: It is up to the user to choose the programming language(s) for implementing the low-level components in a system described by AADL. In some cases all of the components may be implemented in the same programming language (e.g., Ada), while other cases may employ a mixture of programming languages (e.g., some in C and others in Ada). This may pose a challenge for some tools that attempt to transform AADL specifications into glue code. Nevertheless, the AADL needs to remain implementation-neutral with respect to syntax and, as much as possible, semantics.

4.1.12 AADL shall support a means of separating application architecture from execution platform architecture, and supply a flexible means of binding one to the other.

Discussion: The purpose of this requirement is to minimize the dependency of the application on the execution platform. We want to be able to change a processor without requiring changes to the software architecture.

4.1.13 The AADL shall support strong typing of interfaces.

Discussion: Many of the programming languages used in avionics systems employ strong typing, that is, data types can be built up in layers with a set of primitives and type constructors, and the compiler uses all of the type information to check for appropriate use of data elements. AADL must be able to accurately describe the interfaces to such components, though it need not require such detail for a specification to have meaning. The AADL needs to accommodate types characterized by class inheritance in an implementation language.

4.2 Support for Real-Time Applications

4.2.1 The AADL shall support the specification of time properties for components that enable the automated analysis, construction, execution and enforcement of a real-time schedule. These properties shall include execution time required, invocation rate, and alternative execution paths.

Discussion: Time properties are needed to support the automated development of a real-time schedule for the application. By including additional data about nominal execution times, automated analyses can yield information about margins and sensitivity.

4.2.2 The AADL shall support mixed time-driven and event-driven computations with both hard and soft deadlines.

Discussion: Time-driven computations use a periodic clock to trigger computation and communication. Event-driven systems use availability of data and other events to trigger computations. Many applications include both kinds of computation.

Systems will be designed so that only the more critical functions (with hard deadlines) are guaranteed to be schedulable under worst-case execution time bounds. Load shedding of the less critical tasks (with soft deadlines) will occur during intervals of transient processor overload.

4.2.3 The AADL shall support the definition of events that include source and destination components.

Discussion: Certain application components may require knowledge of the origin of the event that triggered their execution. Some events may be predefined from the execution platform while others may be application-specific.

4.2.4 The AADL shall support the association of data with events.

Discussion: Some avionics applications are partially (if not wholly) driven by data availability. This is most naturally expressed as an event that carries the relevant data (or, if you prefer, data whose appearance or update constitutes an event).

4.2.5 AADL shall support specifications from which code can be generated that runs on a POSIX-conformant operating system.

Discussion: Vendor support for the POSIX operating system standards is growing, and real-time extensions to the standard are emerging. The AADL needs to be compatible with this ongoing work. We can only require however, that the AADL is compliant with actual published standards, since there is no way to determine precisely what will happen with drafts. Relevant standards are IEEE 1003.1d, 1003.1j, 1003.1q, 1003.1h, 1003.1m and 1003.13.

4.3 Embedding

4.3.1 The AADL shall support the specification of systems with stringent size, weight, power and efficiency restrictions. It shall be possible to implement simple specifications with little or no run-time environment.

Discussion: Avionics computing hardware is ideally as small and light as possible though often highly complex. This is due in part to the wide variety of vehicles now being produced, including Unmanned Air Vehicles and missiles. AADL must avoid assuming that there will be sufficient processing power to handle integration overheads. The AADL shall be able to represent execution platform resource capacities available to the application components that are relevant to analysis and code generation.

4.3.2 The AADL shall be able to represent execution platform resource capacities available to the application components that are relevant to analysis and code generation.

Discussion: Requirement 4.1.1 called for the inclusion in AADL of both software and hardware component interfaces. This requirement adds some detail to the hardware

components. One of the forms of analysis will be to map application software components onto execution platform components (e.g., a process thread onto a processor, a message onto a channel), so the capacity of an execution platform component is part of its specification.

4.3.3 The AADL shall support the specification of execution platform-specific component types.

Discussion: Avionics devices are often custom or require application specific interfacing, such as digital signal processors, reflective memory, encoder/decoder. It is impossible to anticipate the full range of devices required for all avionics applications.

4.4 Life Cycle Support (of an application)

4.4.1 The AADL shall support incremental refinement and analysis of partial specifications.

Discussion: For example, partition impact analysis could be done without any time attributes filled in; only the interconnection topology is required. Similarly, safety and reliability attributes would not be needed for schedule analysis. Approximate schedule analysis can also be done with a subset of the timing information filled in (e.g., nominal execution times but no worst-case execution times). Coarse definition of thread execution times can be used early on to check feasibility, before detailed information is available about the various execution paths within a thread.

4.4.2 The AADL shall not assume any specific system/software development process.

Discussion: The AADL fits most naturally within the context of model-based development, where the architecture specification is used throughout the lifecycle to explore options, partition functionality, generate documentation and “glue” code, and maintain the system. It must not dictate a particular system/software development process. Rather it needs to fit in with a wide range of process types, including waterfall, spiral, rapid application development, clean room, etc.

4.4.3 The AADL shall support traceability between AADL specifications and related work products.

Discussion: Examples of related work products are source code modules, analysis results, requirements. (See figure 4.) This is important for many organizations developing embedded software, especially those with safety requirements (e.g., commercial avionics).

4.4.4 The AADL shall accommodate the management of configurations of multiple versions of components as well as multiple versions of an architectural specification.

Discussion: Configuration management is a critical facet of the development process for large systems. AADL integrates components that come from a variety of sources and therefore directly overlaps with configuration management concerns.

4.5 Extensibility

4.5.1 The AADL shall provide an extension mechanism that preserves the ability for all standard-compliant tools to read a specification with extensions. The extension mechanism shall use implementation-neutral syntax.

Discussion: One way to keep a standard alive is to make it extensible. The methods for extension must themselves be part of the standard otherwise the market becomes fragmented and tools developed for one “flavor” don’t work with another. Adding attributes is likely to be a common way of extending AADL, since that is where most of the information used by analysis routines is represented. More sophisticated extensions may introduce “sub-language” syntaxes which would require features in the core AADL that cue a parser about how to parse it or to ignore it. Implementation neutrality is required for the extension mechanism, but AADL users may define implementation-specific extensions to avoid prohibitive costs.

Requirement 4.1.3 forbids the AADL to require the specification of component implementation details, but that is not the same as forbidding such details altogether. Some

toolsets may provide value in certain domains by supporting key implementation (e.g., algorithmic) details that enhance analysis, code generation and/or configuration management.

4.5.2 The extension mechanism of AADL shall support the distinction of extensions from different sources.

Discussion: By putting extension mechanisms in the hands of AADL users and tool developers, we open the door to the parallel creation of multiple extensions that serve similar purposes. We do not want to create a central registration organization that arbitrates the extensions, since that would tend to stifle this emerging technology. A better alternative is to provide a way for extension authors to add identifying information that gives them a reasonable assurance that theirs will not be confused with anyone else's.

4.5.3 The AADL syntax shall be structured so that it is straightforward for tools to automatically generate AADL specifications and parse them.

Discussion: This seems like a trivial requirement, but some syntaxes are much easier to automatically generate and parse than others. Parsers are usually built using automated tools and are capable of looking ahead only one token (word or symbol) to determine the interpretation as they progress through the stream of bytes.

5 Annex Requirements

In the same vein as Ada and POSIX annexes, the following requirements are optional for the AADL, but deal with issues that are common enough in the market to warrant standardization. A variant of AADL that addresses any of the requirements in this section must be downward compatible with the core AADL. For example, if an AADL variant includes features that support reliability and fault-tolerance analysis, those features must be added in a way that preserves the core semantics of the language and can be easily and safely ignored by tools that encompass only a core AADL.

The annex requirements are divided into the following categories:

- Reliability and Fault Tolerance—Support for modeling and analyzing these aspects of a system.
- Verification (Assurance of Correctness)—Support for verification in general, and safety in particular.
- Graphical Syntax for AADL—Support for another view of an AADL description.

5.1 Reliability and Fault Tolerance (RFT)

5.1.1 The AADL shall be able to describe fault and error models and assumptions.

Discussion: Fault and error models are the basis for performing reliability analysis. Special syntax could be provided for describing stochastic state-transition models for both software and hardware objects. These error models will define types of faults, errors and error behaviors.

5.1.2 The AADL shall be able to specify reliability requirements for a system.

Discussion: Reliability modeling and analysis allow the system architect to determine the probability of failure of a fault-tolerant system that is subject to randomly arriving fault events.

5.2 Verification (Assurance of Correctness)

5.2.1 The AADL shall support verifying that properties of a specific architecture conform to a more general architecture.

Discussion: A reference architecture is often used to state a set of constraints on the behavior and structure that all systems described by that architecture must satisfy. During the refinement process, the constraints serve as a way to assess conformance. This requirement ensures that AADL may be used in an analogous way.

5.2.2 The AADL shall support the description of multi-level safety and security requirements. This includes attributes that enable analysis to ensure that components do not interact in ways that violate a set of safety and security policies.

Discussion: An example would be to check for compliance with DO-178B safety requirements (reference 1)

5.3 Graphical Syntax for AADL

5.3.1 The AADL graphical syntax and semantics shall be isomorphic to the textual syntax and semantics.

Discussion: Users get frustrated when hard-won information is lost through a translation step.