

The SAE Architecture Analysis & Design Language (AADL) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering

Peter H. Feiler (Software Engineering Institute) SAE AS-2C AADL Standard Co-author
phf@sei.cmu.edu

Bruce Lewis (US Army AMCOM) SAE AS-2C AADL Chair
bruce.lewis@sed.redstone.army.mil

Steve Vestal (Honeywell) SAE AS-2C AADL Standard Co-author
Steve.Vestal@honeywell.com

www.aadl.info

info@aadl.info

Abstract.

Architecture Description Languages provide significant opportunity for the incorporation of formal methods and engineering models into the analysis of software and system architectures. A standard is being developed for embedded real-time safety critical systems which will support the use of various formal approaches to analyze the impact of the composition of systems from hardware and software and which will allow the generation of system glue code with the performance qualities predicted. The SAE AADL standard (International Society for Automotive Engineers (SAE) Architecture Analysis & Design Language) is based on the MetaH language developed under DARPA and US Army funding and on the model driven architectural based approach demonstrated with this technology over the last 12 years. The SAE AADL standard is aimed at supporting avionics, space, automotive, robotics and other real-time concurrent processing domains including safety critical applications.

Introduction

The International Society for Automotive Engineers (SAE) Architecture Analysis & Design Language (AADL) is a textual and graphical language used to design and analyze the software and hardware architecture of real-time systems and their performance-critical characteristics. It is aimed at supporting the avionics, aerospace, and automotive industry. The language is used to describe the structure of such systems as an assembly of software components mapped onto an execution platform. The language can describe functional interfaces to components (such as data inputs and outputs) and performance-critical aspects of components (such as timing). The language can describe how components interact, such as how data inputs and outputs are connected or how application software components are allocated to execution platform components. The language can also describe the dynamic behavior of the runtime architecture by supporting the modeling concept of operational modes and mode transitions. The language is designed to be extensible to accommodate analyses of the runtime architectures that the core language does not completely support. Extensions can take the form of new properties and analysis specific notations that can be associated with components.

The AADL was developed under the auspices of the International Society for Automotive Engineers (SAE). The AADL is developed for embedded systems that have challenging resource (size, weight, power) constraints, that have challenging and strict real-time response requirements that must tolerate faults, that have specialized input/output hardware, and that must be certified to high levels of assurance. Intended fields of application are avionics systems, flight

management, engine and power train control systems, certain medical devices, industrial process control equipment, and space applications. Since the AADL is an architecture description language (ADL) it addresses system of systems issues, where these systems include embedded systems components.

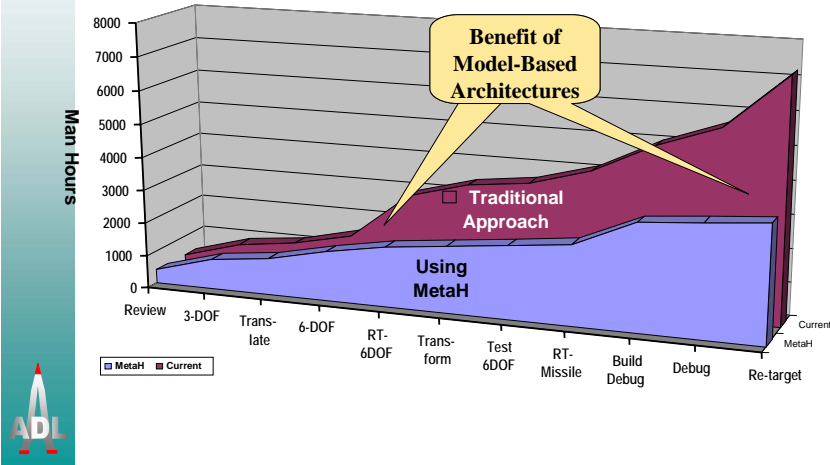
The language can describe important performance-critical aspects such as timing requirements, fault and error behaviors, time and space partitioning, and safety and certification properties. Such a description allows a system designer to perform analyses of the composed components and systems such as system schedulability, sizing analysis, and safety analysis. From these analyses, the designer can evaluate architectural tradeoffs and changes. Since the AADL supports multiple and extensible analysis approaches, it provides the ability to analyze the cross cutting impacts of change in the architecture in one specification using multiple analysis tools. The AADL specification language has been designed to be further used with proper tool support to generate the code needed to integrate the system components and build a system executive. Since the models and the architecture specification drive the design and implementation, they can be maintained to permit model driven architecture based changes throughout the system lifecycle.

Background

The AADL is based on experiences in the use of DARPA funded ADL efforts, in particular MetaH developed by Honeywell [1]. A number of organizations have used MetaH in prototypical system developments, including Boeing, US Army, and the SEI. The case study of a pilot application of the MetaH technology by the U.S. Army AMCOM SED laboratory to missile guidance systems produced some insights into the potential cost savings of an architecture-driven approach. An existing missile guidance system, implemented in Jovial, was reengineered to run on a new hardware platform and to fit into generic missile reference architecture [3]. As part of the reengineering effort the system was modularized and translated into Ada95. The task architecture consisting of 12-16 concurrent tasks was represented as a MetaH model and the implementation generated automatically from the MetaH model and the Ada95 coded application components. The resulting system consisted of 12,000 source lines of application component code, 3000 lines of MetaH executive generated from the MetaH model, and 3000 lines of code representing MetaH kernel services. The engineers doing the reengineering work made a conservative estimate of effort required to reengineer the system into a pure Ada95 implementation and validated the estimate with the prime contractor who implemented the missile. The cost savings ranged from 50% for reengineering to a different language and platform, while a platform port had a cost savings of 90%.

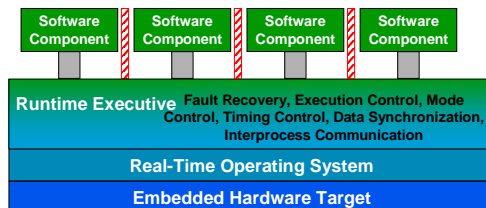
AMCOM Effort Saved Using MetaH

total project savings 50%, re-target savings 90%



MetaH demonstrated the practicality of using an ADL as core modeling notation for providing analysis capabilities of several performance-critical quality attribute dimensions such as schedulability, dependability, and safety-critical concerns. The MetaH toolset demonstrated the capability of not only supporting system analysis, but also automatic generation of glue code in form of a system executive that performs all task binding, dispatching, and inter-task communication with application components as “plug-ins” into this infrastructure. This separation of concerns allows application developers to focus on domain functionality, while a software system architect can focus on achieving system-level performance-critical quality attributes.

Generated Partitioned Architecture



Strong Partitioning

- Timing Protection
- OS Call Restrictions
- Memory Protection

Portability

- Application Components
- Tailored Runtime Executive
- Standard RTOS API

An Analyzable Software System Architecture Description Notation

The AADL has been designed to be a basis for model-based analysis and generation of embedded systems, i.e., embedded and system of systems engineering driven by an architecture that is reflected in the models and maintained throughout the system life cycle [2]. The notation

has been designed as an extensible core language with well defined semantics and both a graphical and textual presentation. The core language supports modeling in several architecture views [7] and addresses timing and performance analyses through explicit modeling of application system and execution platform components and their binding as well as precisely defined concurrency and interaction semantics and timing/performance properties. The extension mechanisms permit properties to be introduced that are specific to additional architecture analyses in terms of other quality attributes such as reliability, security, etc. In this section we introduce to core language and in the next section we discuss the extension capability.

The focus of the AADL is to model the software system architecture in terms of an application system bound to an execution platform. The architecture is modeled in terms of hierarchies of components, whose interaction is represented by connections. Components have a component type that represents its externally visible interface and other characteristics, i.e., represents a component specification, and one or more implementations. A component implementation in the AADL may represent application source text and may be decomposed into an interconnected set of subcomponents that are instances of other component types and implementations. Generalization of components is supported in that component types and implementations can be expressed as extensions of other component types and implementations.

To support modeling of execution platforms four categories of components have been introduced: *processor* as a virtual machine that schedules and executes units of concurrent execution (threads) according to a specified scheduling protocol and may support space partitioning through protected address spaces; *memory* as a storage abstraction that can hold data and/or code; *bus* as a connector abstraction between execution platform components, and a *device* as an abstraction of an active component that an application system can interact with and a processor executing software may require access to via a bus. The execution platform components may represent hardware components or abstract execution platform components, whose implementations may represent virtual machines that are implemented in terms of another execution platform, with the bindings finally resolving to actual hardware. Each execution platform category has a number of predefined properties such as thread and process swap time or scheduling protocol for processors. The core AADL predefines such properties and an initial set of acceptable property value that can be extended. For example, new scheduling protocols can be introduced through a property extension mechanism.

Application system modeling is supported through two groups of component categories. The first group focuses on the runtime behavior of a system and consists of: *thread* as basic unit of concurrent execution; and *process* as unit of protected address space. Threads are contained in processes and have one of a set of predefined dispatch protocol property values or one introduced through the property extension mechanism. Predefined dispatch protocols include periodic, aperiodic, sporadic, and background. Threads have separate execution entrypoints into their associated source text for initialization, nominal execution, and recovery. In case of nominal execution server threads may have multiple entrypoints defined as server subprogram entrypoints. The process load, thread dispatch and scheduling semantics are defined using a hybrid automaton notation.

The second group focuses on the source text of a system and consists of: *package* as unit of source text; and *data* component as passive application data. The package category allows the modeler to represent the source text decomposition structure to a level of detail that is appropriate to the modeling effort. The data component category supports representing data types and class abstractions in the source text as necessary for architecture models. The data type is used to type ports (see below), to specify subprogram parameter types, and to type data component instances. The component type extension mechanism can model type inheritance. Subprogram features (see below) in component types can represent class methods and accessors of data component declared as sharable with a specified concurrency control protocol. Required access to sharable data component instances is specified in a requires subclause of a component type.

A final component category supports hierarchical composition and consists of: *system* as a unit whose implementations can contain execution platform components, application system components and other system instances.

The AADL supports modeling of three kinds of interactions between components: directional flow of data and/or control through data, event, and event data port connections; call/return interaction on subprogram entrypoints; and through access to a shared data component (see data component above).

Threads, processors, and devices, and their enclosing components (process and system) have in ports and out ports declared. Data ports communicate unqueued state data, event ports communicate events that are raised in their implementation, their associated source text, or actual hardware, and event data ports represent queued data whose arrival can have event semantics. Arrival of an event at a thread results in the dispatch of that thread – with semantics defined via property values and hybrid automata for event arrival while the thread is active. For data port connections data is communicated upon execution completion (immediate connection with the effect of mid-frame communication for periodic threads) or upon thread deadline (delayed connection with the effect of phase delay for periodic threads).

The data and event data ports appear to the application source text as data variables – in ports as data variables where input is found when a thread is dispatched, and out ports as variables into which output to be communicated to other components is placed for transfer at well-defined points. In other words, the application source text of a component has no knowledge of the components it interacts with. The interaction connection is defined as part of the AADL description, and appropriate runtime executive code can be generated for thread dispatching and communication.

Subprogram entrypoints are defined in component types as provided and required entrypoints. At the level of components representing source text they represent procedures/functions that are called sequentially. At the level of concurrent components they represent synchronous call/return between two concurrency units (client subprogram calling a server subprogram).

Components can have modes. Modes represent alternative configurations of the component implementation with only one mode being active at a time. At the level of system and process a mode represents possibly overlapping (sub-)sets of active threads and port connections, and alternative configurations of execution platform components, as well as alternative bindings of application components to execution platform components. Mode change behavior is specified as a state transition diagram whose states are the modes and the transitions are triggered by events. Thus, the AADL can model dynamically changing behavior of statically known thread and port communication topologies bound to statically known execution platform topologies. Modes can also be declared for sources text components. This permits mode-specific property values to be declared in situations where the thread and connection architecture does not change, but the thread internal behavior changes, e.g., it has different worst-case execution times under different modes. Such more detailed modeling of application systems allows for less conservative analysis such as schedulability analysis.

The AADL has the following basic fault handling model. Runtime faults may be handled within source text components through mechanisms that are part of the source language runtime environment. For fault not handled at that level or propagated by the source text a thread is given an opportunity to recover and continue with the next dispatch through a recovery entrypoint. Thread unrecoverable errors are propagated as error events. The modeler of a particular application system indicates through event connections where the error event is propagated to, and mode change behavior descriptions indicate actions taken in response to error events.

The AADL also supports other behavior specifications. It supports specification of sequential execution paths within threads to represent control flow within a thread in more detail. It supports

specification of expected invocation patterns on subprogram entrypoints that can be checked against actual invocations. It supports specification of expected event port trigger patterns for a port collection, i.e., a lower-level control flow protocol represented by a collection of event port connections that externally is viewed as a single event connection.

In summary, the core AADL supports modeling of application systems and execution platforms as interacting components with specific semantics and bindings. Such systems are configurable in that components have multiple implementations. Semantics defined as part of the component categories and their predefined properties address timing and resource consumption as well as interaction consistency in terms of matching port types and data communicated through the ports. Behavior descriptions allow for model checking of behaviors as well as mode(state)-specific analyses with less conservative results. The core language does not provide properties and semantics for all possible architecture analyses. Instead the AADL has been made extensible both in terms of language notation and in terms of standard annexes to accommodate further analyses. Annexes that are part of the initial core standard include language extensions such as ability to provide an error model for reliability analysis, and flow specification to support end-to-end flow analysis. Other annexes include a UML profile, an XML interchange format, and an Ada compliance and API annex.

An Extensible Software System Architecture Description Notation

The AADL has been made extensible in three respects. First, modelers can define an extensible set of component specifications in form of component types and implementations by making use of the extension mechanism discussed in the previous section. Second, the language itself can be extended through the ability to introduce new properties and extend the set of valid property values for existing properties. Third, the AADL draft standard includes its specification as a UML profile.

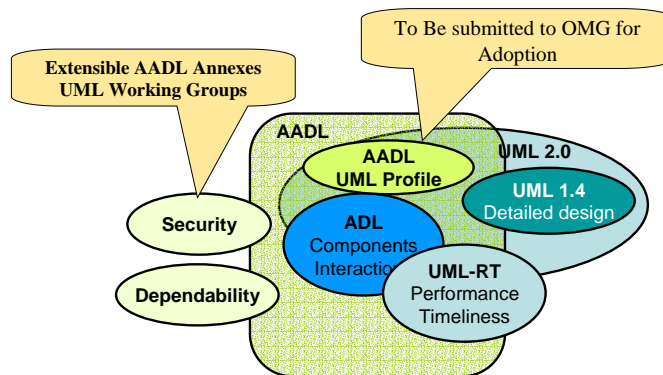
The AADL provides a library concept for organizing component type and implementation declarations. It provides a name scope, thus, facilitates independent development of major subsystems. Furthermore, the component extension mechanism allows modelers to define components and generalizations and specializations of other components.

The AADL currently supports the introduction of new properties extend the set of valid values, and associate them with existing component categories, ports, and connections through property extension sets. No specific notational capability is provided as part of the AADL to describe the semantic meaning of such properties, e.g., in terms of reliability characteristics. Instead, providers of such extension sets can use notations such as the hybrid automaton notation used in the definition of the core language, or resort to English text or other more precise notations to describe the formal model underlying a particular analysis to which the properties represent input.

In many cases it is desirable to express constraints on properties – such as a constraint that the sum of mass property values of any subcomponent with a mass does not exceed a certain maximum. We could consider extending the AADL to explicitly support a constraint language. At this time constraints can be introduced through properties with string values, whose meaning is only understood by constraint analysis tools.

The AADL can be viewed as a modeling notation that can be completed with notations tailored to the specific goals of a particular modeling view and analysis. Such complementary notations can be introduced through string-values properties as suggested above. Alternatively, additional modeling views and semantics addressing certain analyses can be expressed in terms of a UML sublanguage model. This approach is possible because we have developed a UML profile of the AADL as part of the draft standard.

AADL/UML Strategy



Status of the AADL as a Standard

The AADL standard has been in the works since 1999 with a balloted requirements document in 2000. The draft standard has reached a level of maturity that it is in ballot in the Spring of 2004. The specification of the AADL has been aligned with the OMG UML standard to benefit from its large practitioner base. The emerging UML2.0 standard is considered a partner in crime rather than competition. The AADL draft standard includes a UML profile of the AADL, which after being approved as part of the SAE standard, will be submitted for acceptance to the OMG to be part of their standard suite.

The standard provides a means for the commercial production of tools with a common AADL language interface. The UML profile, a specialization providing AADL semantics, will allow the application of formal analysis and code generation tools through a UML graphical specification, enabling the use of currently available UML tools for specification. The UML profile is being developed in parallel with the AADL standard and will be provided as an appendix. We also plan to provide an XML specification for the AADL language once the first version of the language standard is completed. These capabilities will provide an early interface for developing new analysis approaches.

The AADL Standardization Subcommittee also has a liaison relationship with a French research consortium, COTRE, headed by Airbus. COTRE has adopted the AADL for research into new tools, development and analysis methods to support aviation system development requirements. The AADL plays a significant role in a future software and systems development approach described by Airbus and COTRE in a recent paper[4]. Other US and European companies and agencies are evaluating and experimenting with MetaH.

Architecture based, model driven approaches are also beginning to appear in the general software engineering domain. UML 2.0, the Model Driven Architectures Initiative [5], will provide a new layer to UML to directly support a generalized Model driven architecture based approach. It is expected that multiple profiles for different domains will be defined as specializations of UML 2.0. UML 2.0 is expected to be released in mid 2003. The AADL UML profile will incorporate new architecture description capabilities from UML 2.0 when it is released.

The Model Driven Architecture (MDA) Initiative

- Based on the success of UML, the OMG has formulated a vision of a method of software development based on the use of models
- Key characteristic of MDA:
 - The focus and principal products of software development are models rather than programs
 - “The design is the implementation” (i.e. UML as both a modeling and an implementation language)
- UML plays a crucial role in MDA
 - Automatic code generation from UML models
 - Executable UML models
 - Requires a more precise definition of the semantics of UML (UML 2.0)

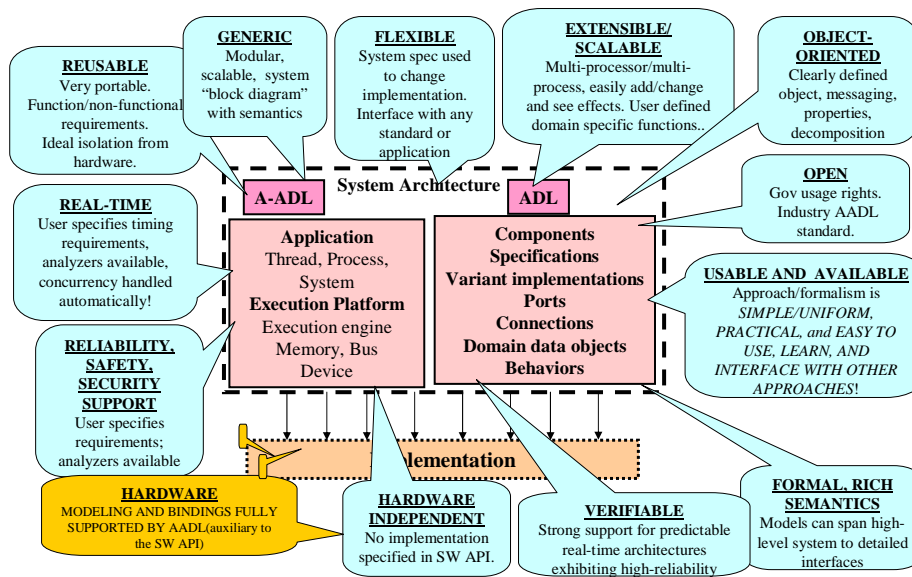
Source: Bran Selic, Rational

The University of Southern California, Center for Software Engineering, lead by Barry Boehm, has announced the development of Model-Based Architecting and Software Engineering (MBASE) approach [6]. This approach currently is being developed to be compatible with several Architecture Description Languages, one being the AADL.

Summary

The AADL has been designed to specifically support the development of large-scale systems through model-based architecture-driven software systems engineering by providing an analyzable architecture description language with well defined semantics. Its roots are in more than a decade's research in architecture description languages with an emphasis on concepts that address performance-critical embedded systems concerns, in particular timing and performance. The standard has been made extensible to permit inclusion of other performance-critical quality attribute concerns through annexes, without bloating the core standard. This permits new analyses to be supported in the future as they emerge from research, e.g., in the area of network security and intrusion management.

The AADL in a Nutshell



The SAE AADL provides an opportunity for the embedded real-time systems research community to have a direct impact on the practitioner community. As the AADL becomes the accepted means for modeling, analyzing, and integrating systems based on architectural models, it can become a vehicle for accelerated transition of research results in new analysis techniques by demonstrating the use of research theories in the context of the AADL/UML.

References

1. Pam Binns, Matt Englehart, Mike Jackson and Steve Vestal, "Domain Specific Software Architectures for Guidance, Navigation and Control," Honeywell Technology Center, Minneapolis, MN, International Journal of Software Engineering and Knowledge Engineering, Vol6, No. 2, 1996, pages 201-227.
2. Peter H. Feiler, Bruce Lewis, Steve Vestal, "Improving Predictability in Embedded Real-time Systems," Carnegie Mellon Software Engineering Institute, CMU/SEI-2000-SR-011, October 2000.
3. David J. McConnell, Bruce Lewis and Lisa Gray, "Reengineering a Single Threaded Embedded Missile application onto a Parallel Processing Platform using MetaH," 5th Workshop on Parallel and Distributed Real Time Systems, 1996.
4. Patrick Farail, Pierre Dissaux, "COTRE a Software Design Workshop", DASIA 2002, May 2002.
5. Bran Selic, "Performance Oriented UML", Tutorial, 3rd International Workshop On Software and Performance, July 2002.
6. Barry Boehm, Overview, Mini Tutorial, <http://sunset.usc.edu/research/MBASE/>
7. Paul Clements, et.al., "Documenting Software Architectures: Views and Beyond", Addison-Wesley, SEI Series in Software Engineering, 2002.