

The SAE Architecture Analysis & Design Language (AADL) Standard

Abstract: The Society of Automotive Engineers (SAE) Architecture Analysis & Design Language (AADL) was developed to support quantitative analysis of the runtime architecture of the embedded software system in computer systems with multiple critical operational properties, such as responsiveness, safety-criticality, security, and reliability by allowing a model of the system to be annotated with information relevant to each of these quality concerns and AADL to be extended with analysis-specific properties. It supports modeling of the embedded software runtime architecture, the computer system hardware, and the interface to the physical environment of embedded computer systems and system of systems. It was designed to support a full Model Based Engineering lifecycle including system specification, analysis, system tuning, integration, and upgrade by supporting modeling and analysis at multiple levels of fidelity. A system can be automatically integrated from AADL models when fully specified and when source code is provided for the software components. See www.aadl.info.

1 Model-based Engineering of Embedded Systems

Safety-critical embedded systems must be developed with a rigorous engineering process in order to meet their requirements. Model-based engineering provides increased predictability in system integration through analysis of architecture models of systems early and throughout the development life cycle. Typically system engineers and control engineers drive the development of such systems. As embedded systems have become increasingly software-intensive and increasingly utilize commercial computing hardware integration of the embedded software system has become a major engineering challenge. The impact of decisions made by software engineers is often not well-understood and violates assumptions made by system and control engineers.

Therefore, a key to engineering an embedded software-intensive system is to analyze the operational characteristics of its realization through a model of its software runtime architecture, its compute platform, its interface to the physical environment, and the deployment of the software on the hardware platform. In order to achieve these operational characteristics, the architecture must provide valid data, at the right time, within resource limitations, with proper security, and provide required levels of fault tolerance and dependability.

Model-based engineering involves the creation of models of the system at a level of abstraction that is relevant to the analysis to be achieved. Traditionally, this has led to the creation of different models of the same system by different stakeholders. Dependability engineering resulted in the creation of fault trees for fault analysis and Markov models for reliability analysis. Resource utilization analysis is based on resource demands and resource capacities. Scheduling analysis is based on a timing model of the application tasks. Security analysis involves the creation of a model in terms of security levels and domains applied to subjects and objects.

In other words, a number of independently created models reflect the same system architecture and any change to this architecture during its life time requires each of these models to be updated and validated that it correctly reflects the actual system architecture. Similarly, it is difficult to consistently reflect any changes in one analysis dimension in other dimensions, e.g., consistently reflect the impact of choosing a different security encryption scheme on intrusion resistance as well as on schedulability and end-to-end latency. So it becomes important to integrate the different analysis dimensions into a single architecture model.

An architecture model that is annotated with analysis-specific information can drive model-based engineering by generating the analysis-specific models from this annotated model. This allows changes to the architecture to be reflected in the various analysis models with little effort by regenerating them from the architecture model. This approach also allows us to evaluate the impact across multiple analysis dimensions. The SAE Architecture Analysis & Design Language (referred to in this document as AADL) was developed for just such a purpose.

2 The SAE AADL Standard

The purpose of the SAE AADL standard is to provide a standard and sufficiently precise (machine-processable) way of modeling the architecture of an embedded, real-time system, such as an avionics system or automotive control system, to permit analysis of its properties, and to support the predictable integration of its implementation. Defining a standard way to describe system components, interfaces, and assemblies of components facilitates the exchange of engineering data between the multiple organizations and technical disciplines that are invariably involved in an embedded real-time system development effort. A precise and machine-processable way to describe conceptual and runtime architectures provides a framework for system modeling and analysis; facilitates the automation of code generation, system build, and other development activities; and significantly reduces design and implementation defects.

The AADL is a textual and graphical language used to model and analyze the software and hardware architecture of embedded systems. These are systems whose operation strongly depends on meeting non-functional system requirements such as reliability, availability, timing, responsiveness, throughput, safety, and security. The AADL is used to describe the structure of such systems as an assembly of software components mapped onto an execution platform. It can be used to describe functional interfaces to components (such as data inputs and outputs) and performance-critical aspects of components (such as timing). The AADL can also be used to 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 be used to describe the dynamic behavior of the runtime architecture by providing support to model 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 standard defines the core AADL that is designed to be extensible. While the core language provides a number of modeling concepts with precise semantics including the mapping to execution platforms and the specification of execution time behavior, it is not possible to foresee all possible architecture analyses. Extensions to accommodate new analyses and unique hardware attributes take the form of new properties and analysis specific notations that can be associated with components. Users or tool vendors may define extension sets. Extension sets may be proposed for inclusion in this standard. Such extensions will be defined as part of a new Annex appended to the standard.

The standard does not specify how the detailed design or implementation details of software and hardware components are to be specified. AADL may be used in conjunction with existing standard languages in these areas. The AADL describes interfaces and properties of execution platform components including processor, memory, communication channels, and devices interfacing with the external environment. Detailed designs for such hardware components may be specified by associating source text written in a hardware description language such as VHDL¹. The AADL can describe interfaces and properties of application software components implemented in source text, such as threads, processes, and runtime configurations. Detailed designs and implementations of algorithms for such components may be specified by associating source text written in a software programming language such as Ada 2005 or C, or domain-specific modeling languages such as MatLab[®]/Simulink^{®2}.

Rules of conformance are specified between specifications written in the AADL, source text and physical components described by those specifications, and physical systems constructed from those specifications. The AADL is not intended to describe all possible aspects of any possible component or system; selected syntactic and semantic requirements are imposed on components and systems. Many

¹ VHDL is the "Very High Speed IC Hardware Description Language. See IEEE VHDL Analysis and Standardization Group for details and status.

² MatLab and SimuLink are commercial tools available from The MathWorks.

of the attributes of an AADL component are represented in an AADL model as properties of that component. The conformance rules of the language include the characteristics described by these properties as well as the syntactic and semantic requirements imposed on components and systems. Compliance between AADL specifications and items described by specifications is determined through analysis, e.g., by tools for source text processing and system integration.

The standard does not prescribe any particular system integration technologies, such as operating system or middleware application program interfaces or bus technologies or topologies. However, specific system architecture topologies, such as the ARINC 653 RTOS, can be modeled through software and execution platform components. The AADL can be used to describe a variety of hardware architectures and software infrastructures. Integration technologies can be used to implement a specified system. The standard specifies rules of conformance between AADL system architecture specifications and physical systems implemented from those specifications.

The standard was not designed around a particular set of tools. It is anticipated that systems and software tools will be provided to support the use of the AADL.

The SAE AADL standard consists of a suite of documents: the core language standard, the Graphical AADL Notation Annex, the AADL Meta Model and XMI Interchange Formats Annex, Language Compliance and Application Program Interface Annex, the Error Model Annex, the UML Profile for AADL Annex, and the Behavior Modeling Annex. The core language standard was published in Nov 2004 (AS5506). Based on the experience with the original standard a revision of the core language standard is going into ballot in the Spring 2008 as AADL V2. The Graphical AADL Notation Annex, the AADL Meta Model and XMI Interchange Formats Annex, Language Compliance and Application Program Interface Annex, and the Error Model Annex were published in June 2006 (AS5506/1). The Behavior Modeling Annex has been adapted to AADL V2 and is going into ballot at the same time as AADL V2. The UML profile for AADL is part of the Object Management Group (OMG) Modeling and Analysis of Real-Time Embedded systems (MARTE) profile, which will be finalized by July 2008.

The Graphical AADL Notation Annex defines a set of graphical symbols for the graphical AADL notation. These graphical symbols can be used to express relationships between components, features, and connections in an AADL model. Graphical AADL diagrams are legal in accordance with the AADL core standard if the AADL model being presented graphically is legal and if the correct graphical symbols are used. For example, a graphical editor is not permitted to create a connection whose source and destination are not connected. Graphical presentations of AADL models are permitted to show subsets of legal AADL models. For example, property values may be entered through a property sheet or dialog box. The figures in this annex present different views of an AADL model. These views are not prescriptive, but intended to illustrate possible views and layouts.

The AADL Meta Model and Interchange Formats Annex defines the AADL meta model and XML-based interchange formats for AADL models. The AADL meta model defines the structure of AADL models, i.e., an object representation of AADL specifications that corresponds to a semantically decorated abstract syntax tree, as well as an object representation of instantiations of system models from AADL specifications to represent system instances. The object representation of AADL models can be manipulated programmatically through an API. The object representation of AADL models can also be persistently stored as XML documents in a standard interchange format. This permits different tools that support the AADL XML schema or XMI meta model specification to interoperate on AADL models. Both the XML schema and the XMI meta model specification for the AADL are derived from the AADL meta model, thus, the two representations are consistent with the meta model.

The Language Compliance and Application Program Interface Annex defines language-specific rules for source text to be compliant with an architecture specification written in AADL. While the AADL is source text language independent, this annex provides guidelines for users to transition between AADL models and source text written in Ada and source text written in C. This annex recommends the use of an Application Program Interface (API) between the application software and the execution environment to

facilitate the use of mixed language application source code modules in a common execution environment.

The Error Model Annex defines features to enable the specification of redundancy management and risk mitigation methods in an architecture, and enable qualitative and quantitative assessments of system properties such as safety, reliability, integrity, availability, and maintainability. This annex defines a sublanguage that can be used to declare error models within an error annex library and associate them with components in an architecture specification. This annex also defines a sublanguage that may be used within an error annex clause within a core AADL standard implementation declaration.

The Behavior Annex enables the expression of high level composition concepts, such as HRT-HOOD synchronization modes in AADL models through behavioral annotations. These annotations preserve the semantics and rules of use defined in the AADL standard. The extensions are introduced either via new properties declared in a *property set* or via the annex mechanism. The behaviors described in this annex are seen as specifications of the actual behaviors: they can therefore be non-deterministic. They are based on state variables whose evolution is specified by the transitions expressed in the behavioral annex. The data and their types are described in AADL.

3 Summary of the core AADL Concepts

This section is an excerpt from the International Society for Automotive Engineers (SAE) Architecture Analysis & Design Language (AADL) Standard Document. It provides a summary of AADL concepts, structure, and use. The first appearance of a term that has a specific meaning will be italicized.

An AADL specification consists of global AADL declarations and AADL declarations. The global AADL declarations are comprised of *package* specifications that contain globally accessible AADL declarations and *property set* declarations. AADL declarations include *component types*, *component implementations*, *feature group types*, and *annex libraries*. AADL declarations can be declared in packages and are therefore accessible to other packages, or they can be declared directly in an AADL specification and not be accessible to packages. AADL component type and implementation declarations model kinds of physical system components, such as a kind of hardware processor or a software program. This standard defines the following categories of components: *abstract*, *data*, *subprogram*, *subprogram group*, *thread*, *thread group*, *process*, *memory*, *bus*, *virtual bus*, *processor*, *virtual processor*, *device*, and *system*. Abstract is a generic component category that can be refined into any of the other categories. They form the core of the AADL modeling vocabulary.

A component type specifies a functional interface in terms of *features*, *flow specifications*, *modes*, and *properties*. It represents a specification of the component against which other components can operate. Implementations of the component are required to satisfy this specification.

A component implementation specifies an internal structure in terms of *subcomponents*, *connections* between the features of those subcomponents, *flows* across a sequence of subcomponents, *modes* to represent operational states, and *properties*. Unlike many other languages, the AADL allows multiple implementations to be declared with the same functional interface.

Components may be hierarchically decomposed into collections or assemblies of interacting subcomponents. A *subcomponent* declares a component that is contained in another component. A component implementation can contain arrays of subcomponents. A subcomponent names a component type and component implementation to specify an interface and implementation for the subcomponent. Thus, component types and implementations act as *component classifiers*. The hierarchy of a system instance is based upon the set of subcomponents of the top-level system implementation. It is completed by iteratively traversing the tree of the component classifiers specified starting at the top-level system implementation subcomponents.

A feature describes a functional interface of a component through which control and data may be exchanged with other components. Features can be *ports* to support directional flow of control and data,

subprograms to represent synchronous procedure calls, and shared access to data, subprograms, subprogram groups, and bus components. Required subcomponent access specifies the need for a component to access components declared outside the component. Provided subcomponent access specifies that a subcomponent contained in a component is made externally accessible. Ports in an AADL specification may map to a variable in a piece of source code, i.e., a storage location in a physical memory. Features can be grouped together into feature groups.

Subcomponents allow systems to be specified as a static and tree-like containment hierarchy. The AADL also allows components to reference subcomponents that are not contained exclusively in the component. This allows a component to be accessed or used in more than one component. For example, static data items contained in a source text software package and represented in AADL as data components may be used by threads in different processes (whose protected address spaces may otherwise be distinct).

Syntactically the terms component type declaration, component implementation declaration, and subcomponent declaration refer to specific grammar rules for each component category. Semantically, a component may have subcomponents while it itself is a subcomponent of some other component. The terms component and subcomponent must be interpreted semantically as a relationship between two components that are identified by context.

Any named model element, e.g., components, features, modes, connections, flows, and subprogram calls can have properties. A property has a name, a type and a value. Properties are used to represent attributes and other characteristics, such as the period and deadline of threads. When properties are associated with declarations of component types, component implementations, features, subcomponents, connections, flows, and modes, they apply to all respective instances within a system instance. The AADL also supports the specification of instance specific values of any unit in the containment hierarchy of a system instance. AADL tools may record these values for use in the analysis of the system instance or for use in the construction of new system instances. Properties can have mode-specific and binding-specific values.

This standard defines a set of predeclared properties and property types. Additional properties and property types to support new forms of system analysis can be introduced through property sets. Property values can be associated with component types, component implementations, subcomponents, features, connections, flows, modes, and subprogram calls. For example, a property is used to identify the source code files associated with a software component. Another example of the use of properties is specifying hardware memory, i.e., the number of addressable storage units and their size.

Packages provide a library-like structure for organizing component type, component implementation, and feature group type declarations into separate namespaces. A component classifier in a package is referenced by qualifying its name with the package name. Packages can have a nested naming hierarchy.

Features and flow specifications of component types may be partially specified. Similarly, subcomponents, connections, flows, and modes of component implementations may have incomplete specifications. These incomplete component type and implementation declarations act as templates that can be parameterized by specifying *prototypes*. These specifications may be later refined in component type and component implementation extensions with the completion of classifier references and property associations. Component type extensions can also introduce additional features, flow specifications, and properties. Such extensions can add new subcomponents, connections, flows, modes, and properties to component implementations. This allows conceptual and reference architectures to be specified and to be refined into fully specified runtime architectures.

A system modeled in AADL consists of application software mapped to an execution platform. Data, subprograms, subprogram groups, threads, thread groups, and processes collectively represent application software. They are called *software* components. Processor, virtual processor, memory, bus, virtual bus, and device collectively represent the execution platform. They are called *execution platform*

components. Execution platform components support the execution of threads, the storage of data and code, and the communication between threads. Systems are called *compositional* components. They permit software and execution platform components to be organized into hierarchical structures with well-defined interfaces. Operating systems may be represented through properties of the execution platform or, requiring significantly more detail, modeled as software components. In addition, AADL supports modeling in terms of abstract components that can get refined into components of the above categories.

Software components model *source text, virtual address spaces, concurrent tasks* and their interactions. Source text can be written in a programming language such as Ada 2005, C, or Java, or domain-specific modeling languages such as Simulink, SDL, ESTEREL, LUSTRE, and UML, for which executable code may be generated. The source text modeled by a software component may represent a partial application program or model (e.g., they form one or more independent compilation units as defined by the applicable programming language standard). Rules and permissions governing the mapping between AADL specification and source text depend on the applicable programming or modeling language standard. Predeclared component properties identify the source text container and the mapping of AADL concepts to source text declarations and statements. These properties also specify memory and execution times requirements and other known characteristics of the component.

AADL data components represent static data in source text. This data can be shared by threads and processes; they do so by the indicating that they require access to the external data component. Concurrent access to data is managed by the appropriate concurrency control protocol as specified by a property. Realizations of such protocols are documented in an appropriate implementation Annex in this standard.

Data types in the source text are modeled by the declarations: data component type and data component implementation. Thus, a data component classifier represents the data type of data components, ports, and subprogram parameters.

The subprogram component models source text that is executed sequentially. Subprograms are callable from within threads and subprograms. Subprograms may require access to data components and may contain data subcomponents to represent local variables. Subprogram groups represent source code libraries.

AADL thread components model concurrent tasks, i.e., concurrent runtime threads of execution through source text (or more exactly, through binary images produced from the compilation, linking and loading of source text). A scheduler manages the execution of a thread. The dynamic semantics for a thread are defined in this standard using hybrid automata. The threads can be in states such as suspended, ready, and running. State transitions occur as a result of dispatch requests, faults, and runtime service calls. They can also occur if time constraints are exceeded. Error detection and recovery semantics are specified. Dispatch semantics are given for standard dispatch protocols such as periodic, sporadic, and aperiodic threads as well as background threads. Additional dispatch protocols may be defined. Threads can contain subprogram and data components, and provide or require access to data components.

AADL thread groups support structural grouping of threads within a process. A thread group may contain data, thread, and thread group subcomponents. A thread group may require and provide access to data components.

AADL process components model space partitions in terms of virtual address spaces containing source text that forms complete programs as defined in the applicable programming language standard. Access protection of the virtual address space is enforced at runtime if specified by the property `Runtime_Protection`. The binary image produced by compiling and linking this source text must execute properly when loaded into a unique virtual address space. As processes do not represent concurrent tasks, they must contain at least one thread. Processes can contain thread groups, threads, and data components, and can access or share data components.

Execution platform components represent hardware and software that is capable of scheduling threads, of enforcing specified address space protection at runtime, of storing source text code and data, of interfacing with an external environment, and of performing communication for application system connections.

AADL processor components are an abstraction of hardware and software that is responsible for scheduling and executing threads. In other words, a processor may include functionality provided by operating systems. Alternatively, operating systems can be modeled like application components. Processors can contain memory and require access to buses. Processors can support different scheduling protocols. Threads are bound to processors for scheduling and execution.

AADL virtual processors represent virtual machines or hierarchical schedulers. They can be contained in or bound to processors. Processes and threads can be bound to them.

AADL memory components model randomly accessible physical storage such as RAM or ROM. Memories have properties such as the number and size of addressable storage locations. Binary images of source text are bound to memory. Memory can contain nested memory components. Memory components require access to buses.

AADL bus components model communication channels that can exchange control and data between processors, memories, and devices. A bus is typically hardware that supports specific communication protocols, possibly implemented through software. Processors, memories, and devices communicate by accessing a shared bus. Buses can be directly connected to other buses. Logical connections between threads that are bound to different processors transmit their information across buses that provide the physical connection between the processors. Buses can require access to other buses.

AADL virtual buses represent virtual channels or communication protocols that perform transmission within processors or across buses.

AADL device components model physical entities in the external environment, e.g., a GPS system, or entities that interface with an external environment, e.g. sensors and actuators as interface between a physical plant and a control system. Devices may represent a physical entity or its (simulated) software equivalent. They may exhibit simple, e.g., a timer, or complex behaviors, e.g., a camera or GPS. Devices are logically connected to application software components and physically connected to processors. They cannot store nor execute external application software source text themselves, but may include driver software executed on a connected processor. A device requires access to buses.

AADL systems model hierarchical compositions of software and execution platform components. A system may directly or indirectly contain data, subprogram, subprogram groups, thread, thread group, process, memory, processor, virtual processor, bus, virtual bus, device, system as well as abstract subcomponents. A system may require and provide access to data and bus components. Execution platform component can be systems in their own right and be modeled using system implementations. For example, a system implementation can be associated with a device that models a camera. This system implementation describes the internal of the camera in terms of the CCD sensor a device, a DSP processor, a general purpose processor as well a software that implements the image processing and download capability of the camera.

AADL modes represent the operational states of software, execution platform, and compositional components in the modeled physical system. A component can have mode-specific property values. A component can also have mode-specific configurations of different subsets of subcomponents and connections. In other words, a mode change can change the set of active components and connections. Mode transitions model dynamic operational behavior that represents switching between configurations and changes in component-internal characteristics, such as conditional execution source text sequences or operational states of a device, that are reflected in property values. Other examples of mode-specific property values include the period or the worst-case execution time of a thread. A change in operating mode can have the effect of activating and deactivating threads for execution and changing the pattern of

connections between threads. A mode subclause in a component implementation specifies the mode states and mode change behavior in terms of transitions; it specifies the events as transition triggers. Subcomponent and connection declarations as well as property associations declare their applicability (participation) in specific modes.

This standard defines several categories of features: *data port*, *event port*, *event data port*, *feature group*, *subprogram parameter*, and *provided* and *required* subcomponent *access*. Data ports represent connection points for transfer of state data such as sensor data. Event ports represent connection points for transfer of control through raised events that can trigger thread dispatch or mode transition. Event data ports represent connection points for transfer of events with data, i.e., messages that may be queued. Ports groups support grouping of ports, such that they can be connected to other components through a single connection. Data subprograms represent entrypoints to code sequences in source text that are associated with a data type. Subprograms represent connection points for synchronous call/returns between threads; in some instances the call/return may be remote. Subprogram parameters represent in and out parameters of a subprogram. Data component access represents provided and required access to shared data. Bus component access represents provided and required access to buses for processors, memory, and devices. Subprogram component access represents the binding of a subprogram call to the subprogram being called.

AADL connections specify patterns of control and data flow between individual components at runtime. A semantic connection can be made between two threads, between a thread and a device or processor for port connections, between a data component and threads that access the data component for data access connections, between a bus component and buses, memory, processor, and device components for bus access connections, between a subprogram component and threads that require call access to the subprogram, or between the event port of a thread, device, or processor and a mode transition for mode transition connections. A semantic connection is represented by a set of one or more connection declarations that follow the component hierarchy from the ultimate connection source to the ultimate connection destination. For example, in Figure 1 there is a connection declaration from a thread out port in Thread1 to a containing process out port in Process3. This connection is continued with a connection declaration within System1 from Process3's out port to Process4's in port. The connection declaration continues within Process4 to the thread in port contained in Thread2. Collectively, this sequence of connections defines a single semantic connection between Thread1 and Thread2. Threads, processes, systems, and ports are shown in graphical AADL notation.

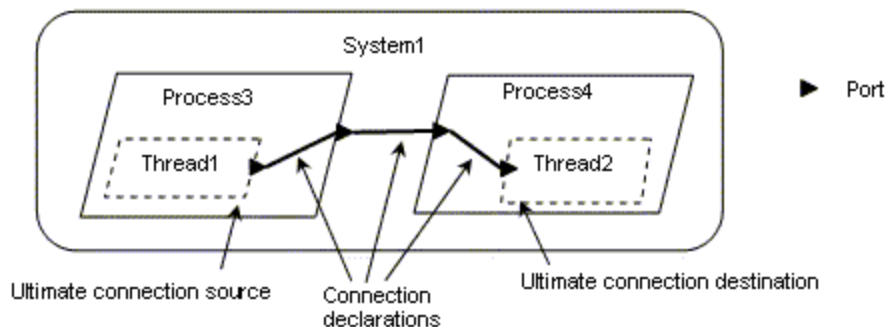


Figure 1 Example Semantic Connections

Flow specifications describe externally observable flow of information in terms of application logic through a component. Such logical flows may be realized through ports and connections of different data types and a combination of data, event, and event data ports, as well as flows through shared data components. Flow specifications represent *flow sources*, i.e., flows originating from within a component, *flow sinks*, i.e., flows ending within a component, and *flow paths*, i.e., flows through a component from its incoming ports to its outgoing ports.

Flows describe actual flow sequences through components and sets of components across one or more connections. They are declared in component implementations. Flow sequences take two forms: *flow*

implementation and *end-to-end flow*. A flow implementation describes how a flow specification of a component is realized in its component implementation. An end-to-end flow specifies a flow that starts within one subcomponent and ends within another subcomponent. Flow specifications, flow implementations, and end-to-end flows can have expected and actual values for flow related properties, e.g., latency or rounding error accumulation.

A physical system is modeled by instantiating a system implementation that consists of subcomponents representing the application software and execution platform components used to execute the application, including devices that interface with the external environment. A system instance represents the complete component hierarchy as specified by the system classifier's subcomponents and the subcomponents of their component classifiers down to the lowest level defined in the architecture specification.

An AADL specification may be used in a variety of ways by a variety of tools during a broad range of life-cycle activities, e.g. for documentation during preliminary specification, for schedulability or reliability analysis during design studies and during verification, for generation of system integration code during implementation. Note that application software components must be bound to execution platform components - ultimately threads to processors and binary images to memory in order for the system to be analyzable for runtime properties and the physical system to be constructed from the AADL specification. Many uses of an AADL specification need not be fully automated, e.g. some implementation steps may be performed by hand.

The AADL core language is extensible through property sets, *annex subclauses* and *annex libraries*. Annex subclauses consist of annex-specific sublanguages whose constructs can be added to component types and component implementations. Annex libraries are declarations of reusable annex-specific sublanguage elements that can be referenced in annex subclauses.

4 Conclusion

The AADL was developed to meet the special needs of performance-critical real-time systems, including embedded real-time systems such as avionics, automotive electronics, or robotics systems. 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 a variety of analysis tools. The AADL specification language is designed to be used with analysis tools that support the automatic generation of the source 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.

See www.aadl.info for a resource on AADL and <http://www.sei.cmu.edu/pcs/> for Model-Based Engineering at the SEI.