

# A Comparison and Evaluation of Real-Time Software Systems Modeling Languages

Kenneth D. Evensen and Dr. Kathryn Anne Weiss

*California Institute of Technology, Jet Propulsion Laboratory, 4800 Oak Grove Dr. Pasadena, CA 91109*

**A model-driven approach to real-time software systems development enables the conceptualization of software, fostering a more thorough understanding of its often complex architecture and behavior while promoting the documentation and analysis of concerns common to real-time embedded systems such as scheduling, resource allocation, and performance. Several modeling languages have been developed to assist in the model-driven software engineering effort for real-time systems, and these languages are beginning to gain traction with practitioners throughout the aerospace industry. This paper presents a survey of several real-time software system modeling languages, namely the Architectural Analysis and Design Language (AADL), the Unified Modeling Language (UML), Systems Modeling Language (SysML), the Modeling and Analysis of Real-Time Embedded Systems (MARTE) UML profile, and the AADL for UML profile. Each language has its advantages and disadvantages, and in order to adequately describe a real-time software system's architecture, a complementary use of multiple languages is almost certainly necessary. This paper aims to explore these languages in the context of understanding the value each brings to the model-driven software engineering effort and to determine if it is feasible and practical to combine aspects of the various modeling languages to achieve more complete coverage in architectural descriptions. To this end, each language is evaluated with respect to a set of criteria such as scope, formalisms, and architectural coverage. An example is used to help illustrate the capabilities of the various languages.**

## I. Introduction

The need to correctly specify a software architecture and design for safety-critical real-time systems given the growing field of application is paramount. A model-driven approach to real-time software development focuses on conceptualizing the software and hardware components, making it easier to address stakeholder concerns early in the development lifecycle. Models can assist in assessing feasibility of implementing both functional and quality attribute requirements. This support is the goal of each of the four modeling techniques presented in this paper, the Architectural Analysis and Design Language (AADL), the Unified Modeling Language (UML), Systems Modeling Language (SysML), the AADL for UML profile, and the Modeling and Analysis of Real-Time Embedded Systems (MARTE) UML profile, but these languages approach this goal with a variety of similarities and differences that will be discussed throughout this paper. First, it is important to understand the general background and originating purpose of each language. Next, each language is evaluated with respect to several dimensions of comparison: scope, formalism, architectural coverage, tool support, and extension mechanisms. A brief model comparison is presented to illustrate the notational differences of these languages and their representation of time. Finally, the limitations of each modeling language are discussed.

## II. Background

The Architecture Analysis and Design Language (AADL) is an SAE published standard. AADL was formerly known as the Avionics Analysis and Design Language and is based upon MetaH (from Honeywell)<sup>1</sup>. AADL is first a language with strict semantics defining an underlying meta-model. It uses a combination of graphical and textual modeling that are closely coupled to one another. Many details, such as user-defined properties, do not appear in the graphical model, but appear strictly in the textual model. AADL is composed of a relatively small set of modeling components to abstract the software and hardware entities found in a real-time embedded system. The manner in

which these components interact is also strictly defined. AADL does not provide a mechanism to change the meaning of its components and connectors. However, AADL does provide mechanisms, such as properties, that enable the modeler to detail the model and bring it closer to the modeler's domain of concern.

AADL descriptions are independent of particular viewpoints; the modeler has control over how entity relationships and architectural concerns are addressed as long as the semantics of the language are observed. Specifically, the modeler can choose which type of AADL components are modeled in each diagram, based on the desired level of abstraction.

Succinctly, AADL captures the execution nature of real-time software and hardware systems. It is often in the execution architecture that the concerns of performance, safety, and reliability are addressed. However, AADL does not address the compile time or explicit deployment nature of the hardware or software. If such views are required, another modeling language, such as the Unified Modeling Language (UML), can be used as part of the design description.

UML was developed to satisfy the need for a common modeling language for object oriented systems and support a widespread application of the design approach. UML 1.0 was submitted to the Object Management Group as a response to an RFP for a standard modeling language, in 1997<sup>3</sup>. UML is now in its second revision. The Systems Model Language (SysML) is a UML profile for systems engineering. However, a more accurate characterization would be that SysML has a strong foundation in UML 2.0, but provides diagrams as well as components beyond UML. SysML provides systems engineers with a notation similar to UML, but also diagrams and tool support specific to the systems engineering level, thereby providing a framework to seamlessly transition between the system engineering level of abstraction in SysML and the architecture and design of the software subsystem in UML.

UML and SysML are diagram centric, notational languages. These notations rely on graphical representations of software components annotated by narrative text. UML and SysML diagrams specify the type of graphical components that are used in a model. Software components modeled graphically in a particular manner in one UML diagram (e.g. a class diagram) may be represented and annotated differently in another diagram.

UML and SysML employ diagrams that follow certain rules defined in their respective specifications to communicate entity relationships and interactions. Diagrams in UML fall into two categories: behavioral (e.g. activity, sequence, and state-machine diagrams) and structural (e.g. class, object, and component diagrams)<sup>3</sup>. As previously stated, UML provides a foundation for SysML, and SysML adds constructs not found in UML such as the requirements diagram and parametric diagram. At the same time, SysML does not use all entities provided in UML as they are not appropriate for the systems engineering domain. Examples include the UML object and communications diagram that speak to software entity relationships<sup>9</sup>.

Architectural views are detailed using the necessary combination of diagrams. The modeler is not restricted to the diagrams presented in the UML or SysML specification. Additional diagram types can be generated in order to detail the design of the software system as needed by the modeler. The adaptability of UML and SysML to the modeler's domain via a profile is a key trait of these languages. Profiles are a combination of user defined meta-models and UML stereotypes that adapt the underlying semantics of UML and SysML to address the needs of the modeler.

In a safety-critical real-time application, there is a strong need for strict semantics in software architecture descriptions to reduce ambiguity in design. It is possible that the flexibility found in UML could be counter to this purpose. One effort to address this concern was the UML Profile for AADL. The UML Profile for AADL adapts the semantics of AADL to UML notation thereby providing a strong semantic structure for specifying real-time systems in UML notation. AADL does not have the capability for describing the dynamic nature of the software system. But, if presented in UML notation, UML behavior diagrams can be used to fill that void in the model specification.

The UML Profile for AADL will be standardized and defined as a part of the Modeling and Analysis of Real-Time Embedded Systems (MARTE) meta-model. In order to assist in bridging between modeling notations, a joint tutorial exists on creating AADL models with MARTE<sup>7</sup>. There is no longer an effort to continue development of

this profile outside of incorporating it into MARTE. Therefore, discussion of the UML Profile for AADL will be limited.

MARTE is a UML profile designed to handle real-time embedded systems and software concepts<sup>7</sup>. UML was simply not designed to address concerns such as scheduling, performance, and time. MARTE's underlying meta-model, based on AADL's meta-model, provides the capability to address these concerns using UML notation. Various components from SysML, such as blocks and ports, are carried over into MARTE.

MARTE does not introduce any new diagrams as part of its meta-model. It does, however, include guidelines for constructing models at various levels of abstraction. High Application Level Modeling, in MARTE, suggests methods for constructing diagrams at a higher-level of abstraction while addressing basic real-time concepts such as schedulability. The Detailed Resource Modeling guidelines, broken into two subsets, namely Hardware Resource Modeling and Software Resource Modeling, addresses hardware execution support and the application programming interfaces of software multi-tasking execution respectively<sup>1</sup>.

### **III. Comparison and Evaluation**

As stated, the dimensions of comparison are scope, formalism, architectural coverage, tool support, extension mechanisms, model notation, and the limitations of each language. Scope addresses the ability to model a real-time software system as opposed to object-oriented or business software. These languages are based on different underlying formalisms, i.e. AADL is a declarative language as opposed to UML which is notation. Architectural coverage addresses the ability of these languages to specify the various aspects of a software system architecture. The discussion on tool support provides an overview of the capabilities tools provide to assist in developing software models using each language. While each language has varying levels of formalism they can all be extended using each language's extension mechanisms. The model comparison provides an example simple software system modeled in UML, AADL, and MARTE and illustrates how each of these languages represents time. Finally, the limitations of these modeling language is provided.

#### **A. Scope**

Each modeling languages focuses on different aspects of the software system. This section on scope addresses the differences in focus of each of these languages.

AADL focuses on the execution entities of a real-time system such as concurrent tasks. Version 1.0 of AADL does not provide the means to specify behavior. The AADL Behavior Annex (annexes are extensions in AADL and will be discussed later) is based on a state-machine meta-model and is supported by version 2.0 of AADL. This standardized annex provides a tightly coupled behavior specification.

AADL models software runtime entities through thread and port connections. Ports and flows are strictly data and event connections as they relate to software. Avionics components, such as processors, memories, and bus, are also modeled. Finally, there is also a notion of modes in AADL for modeling different configurations.

The description of the software execution on a hardware platform forms only a partial description of a real-time system. Often, the complete architectural specification of a real-time system requires some description of the implementation, deployment, and behavior of the software. UML provides the constructs for modeling these aspects of the architecture.

UML can be used to define both a high-level architecture as well as detailed design. UML can describe the execution nature of software components as well as their compile time and deployment characteristics. The execution nature is described with activity and system sequence diagrams, both of which speak to the sequential execution nature of the software. UML can be used to model use cases and user interaction early in the life cycle. Beyond software entities, UML can be used to describe software project organization.

Further augmentation to the architecture description is provided by SysML. SysML addresses concerns such as partitioning various subsystems beyond software, requirements traceability (SysML requirements diagrams), and mapping driving arithmetic equations to components (parametric diagrams). SysML flows can be used to describe

not only the flow of data through a system, but also energy and liquid (for example). Since the semantics of SysML are rooted in UML, the ability to then design a software subsystem in UML is more fluid.

The UML Profile for AADL extends the UML meta-model to describe AADL components and connections. The scope is similar to that of AADL in that the components and connectors map to the elements of a real-time embedded system. There is added benefit in incorporating AADL components into UML diagrams. Because AADL does not provide for modeling the behavior of an executing software system, utilizing the UML profile for AADL makes linking these diagrams to UML sequence diagram, for instance, rather simple.

MARTE contains the constructs necessary to model a high-level architecture as well as the detailed design of both hardware and software components in an embedded real-time system. MARTE has the facilities to model both the software and avionics hardware platform components. Components and connectors are used in describing the relationship in between executing entities using ports and lines. Like UML, MARTE also employs connections to specify the implementation and compile time relationships between software components. MARTE provides entities that can be used to model components of a middleware or real-time operating system such as watchdog timers, schedulers, message-queues, and interrupts. While these concepts can be modeled in AADL, they are more explicit in MARTE.

## **B. Formalism**

Formalism provides the underlying semantics on how to use any language. The modeling languages presented have different underlying formalisms. In some instances, the formalism is stricter than in other instances and focuses on different aspects (diagram centric vs. non-diagram centric). This section compares the languages with respect to these differences.

AADL employs strict component and connector semantics, specifying explicit relations between components. These connections include port connections, shared access connections, and service call connections. Component relationships are fixed but can be augmented by classifier and reference properties as well as annex extensions. An important relationship is the subcomponent relationship. Subcomponents (children components) are defined in an AADL component's implementation. The AADL language defines which component types may be subcomponents of another. For example, AADL threads may be subcomponents of an AADL process but not vice-versa. Figure 1 (Table 13-1 in "The Architecture Analysis & Design Language (AADL): An Introduction") shows which components are allowable subcomponents of other types.

Category Group	Component Category	Permitted Subcomponents	Permitted Subcomponent of
<b>Software</b>	process	thread data thread group	system
	thread	data	process thread group
	data	data	process thread data thread group system
	thread group	data thread thread group	process thread group
	subprogram	None allowed	None
<b>Execution Platform</b>	processor	memory	system
	memory	memory	processor memory system
	bus	None allowed	system
	device	None allowed	system
<b>Composite</b>	system	process data processor memory bus device system	system

**Figure 1 – AADL Component Relationship<sup>5</sup>**

Architectural dynamics (operational modes and dynamic reconfiguration) are expressible via mode-specific active components, connections, and property values. These dynamics are not to be confused with behavior-related modeling. Here, architectural dynamics refer to a modal configuration of components tied to an AADL event port. The state of a system may be inferred from which components are active based on the modal configuration.

In AADL, flow specifications provide for scalable end-to-end flows across component hierarchy. AADL flow specifications define data flow across sibling components. In implementation, flows are detailed across subcomponents. There is a separation between textual and graphical specifications, but the mapping is one-to-one.

There is no way to change the meaning of an AADL component. Properties and Annex Libraries in AADL can be used to detail a component, but do not change the underlying semantics.

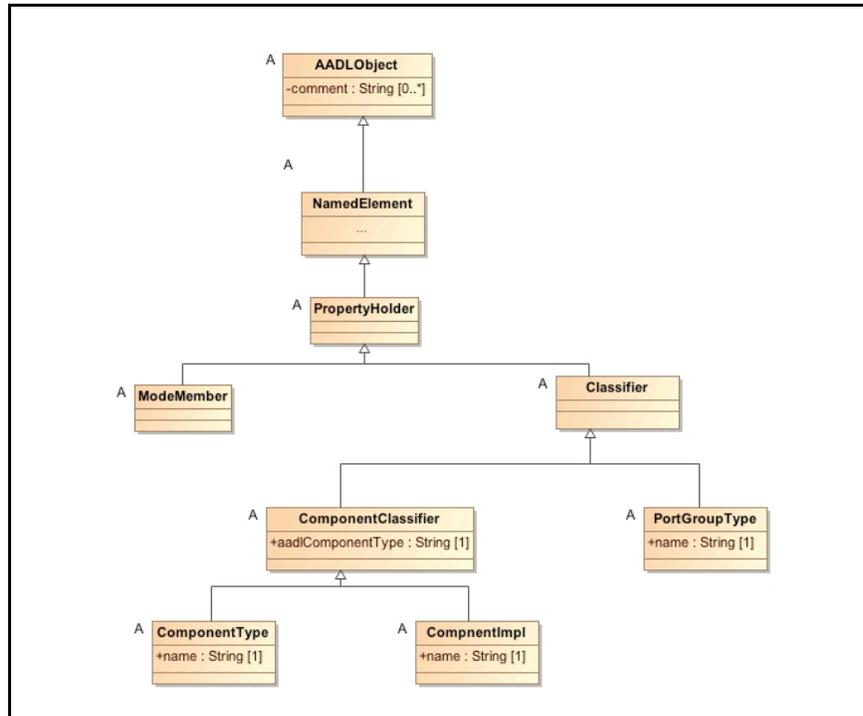
Formalism in UML and SysML stems from the intent of the diagrams used in the design specification. For example, a class diagram shows static relationships among software classes. But in a UML sequence diagram, the intent is to illustrate component interaction, and therefore a different notation is employed. In order to thoroughly describe the architecture or design a system, the appropriate combination of diagrams must be selected.

Diagrams in UML and SysML are heavily graphical in notation. Beyond labeling components, the textual specification is embedded in the graphical model as notes, providing an informal narrative. For example, the notion of “standard ports” in UML is similar to those in SysML representing services required or provided by the entity owning the port. A flow port in SysML does not only apply to data, but also to other entities in the systems engineering domain, such as material or energy<sup>3</sup>. “SysML ‘flow ports’ extend UML 2 ports.”<sup>3</sup>

Meta-models are another method for specifying the underlying relationships of entities and stereotypes in UML diagrams. As of UML 2.0, components, relationships, and even behavior are themselves defined by a meta-model. As stated previously, the meta-model for the UML profile for AADL is derived from the formalisms of the AADL

language. The concepts of AADL, such as packages classifiers, and properties, share a relationship that can be modeled in UML.

Figure 2 depicts a possible relation of AADL concepts based on the AADL meta-model found at <http://aadl.sei.cmu.edu/aadl/currentsite/tool/metamod.html>. It is important to understand that AADL was created first, and then the object representation of the language concepts came later.



**Figure 2 – AADL Class Diagram**

Other diagrams in the meta-model dictate the constraints of the relationship between component classifiers, features (such as ports) and their owning components, and connections. Like UML, the appropriate diagrams dictate the formalism in the UML Profile for AADL.

The UML Profile for AADL helps to set the structure for the MARTE meta-model which in turn defines MARTE’s underlying formalisms. The same diagrams used in UML and SysML can be adapted to MARTE. The MARTE specification provides guidelines for constructing models at various levels of abstraction. These guidelines do not dictate the type of diagrams to use, but suggest which MARTE components and packages to use at conceptual and detailed levels of design.

### **C. Architectural Coverage**

With the exception of SysML, these modeling languages address different aspects of the software system from the conceptual-level of architecture abstraction down to the detailed design of a software system. SysML provides modeling capabilities from the perspective of the system-as-a-whole, i.e. the context within which the embedded system must operate. When examining the breadth of viewpoints at each level of abstraction, AADL is not as capable as other modeling techniques.

AADL components can be used in a variety of combinations to create views that address varying stakeholder concerns. The AADL System component is a high-level container that implies refinement in other models and views. For example, it can be used in a conceptual view and be refined to software (e.g. AADL Thread component) or hardware (e.g. AADL memory component) in other views. The AADL Subprogram component is representative of a function call in an executing system. Parameters passed between functions and the direction of the function call can be detailed in the context of an executing thread. AADL also allows for the mapping of software source code to the executing entities by use of an AADL data subcomponent. However, any conceptual view or detailed design model only speaks to the execution nature of the software; any compile-time or deployment characteristics are not explicitly captured by the semantics of AADL but can be included as a custom property.

AADL enables the modeler to provide a consistent, semantically-strict, static view of the execution software mapped to the avionics hardware. AADL emphasizes a static view, but does not provide the facilities to describe the behavior of real-time software. In order to accomplish this, the AADL model would need to be augmented by an embedded annex library (such as the behavior annex, which is outside the scope of this paper), or a diagram generated in another modeling language such as UML.

UML provides multiple behavior diagrams for characterizing sequential interactions between components at different levels of abstraction. UML also provides diagrams for capturing the implementation (UML Deployment Diagram) in the manner of describing where each executable “lives” on the hardware system. This can be modeled somewhat in AADL by “binding” an AADL process component to an AADL memory component, but the UML deployment diagram describes implementation more explicitly and at a higher level of abstraction.

The UML component diagram is often used to specify the “build” instructions for the software by capturing how source code components are related. This is different than how classes interact with one another, as the conceptual relationship between classes may not necessarily be one to one with the relationship between source code components. The need to model user-interaction with the software is accomplished primarily through UML use-case diagrams. As previously discussed, AADL does not have an explicit means to model a use-case. In general purpose software applications, use-cases often drive requirements.

MARTE, because it is an extension of UML, employs the same types of UML diagrams to gain more thorough architectural coverage. The MARTE specification provides guidelines on which components to use at different levels of abstraction, but does not specify which diagrams are needed to thoroughly describe the architecture - this decision is left to the modeler.

The UML Profile for AADL is similar to AADL in its ability to provide architectural coverage. As the profile’s name suggests, AADL is captured in UML notation. Components modeled in a static diagram, for example a class diagram, can easily be traced to other diagrams, such as a UML sequence diagram.

### **D. Tool Support**

The availability and functionality of tools must be considered when making a decision to utilize one or more software architecture description languages. The languages presented have varying levels of tools support. The tool support described here is not exhaustive, but provides an idea of the availability and usability of tools.

The strongest tool support for performing modeling and analysis in AADL comes from the Software Engineering Institute. The Open Source AADL Tool Environment (OSATE) is built on the Eclipse Rich Client Platform (RCP) and is based on the Topcased project<sup>9</sup>. OSATE provides real-time semantic checking on the AADL model, enforcing semantics early in the model development lifecycle. OSATE also comes with a suite of automated

analysis tools that rely on properties the modeler embeds in the AADL model. Such analyses include schedulability, priority inversion checking, and memory load analysis. Since OSATE is built on the RCP platform, custom analysis plug-ins can be developed to extend the analysis suite as the modeler requires. The most current release of OSATE supports version 2.0 of AADL. Ocarina is another tool environment supporting AADL version 1.0 and 2.0<sup>8</sup>. It uses a command line interface to perform analysis on models. The models need to be generated in a text editor. Ocarina has the ability to generate C source code from these model definitions.

UML enjoys a large suite of tools for generating models. These tools range from free and open-source software, such as Papyrus UML2 Modeler, to commercial software products such as MagicDraw UML (No Magic, Inc.) and Visual Paradigm for UML (Visual Paradigm International). MagicDraw UML and Papyrus UML2 (built on the Eclipse RCP) provide support for modeling in SysML as well as MARTE<sup>10</sup>. While there are many choices, the ability to conduct automated analyses is still lacking in many of these tools. This stems from the fact there is no consistent manner in which to annotate models based in UML, as there are no underlying semantics to do so. MARTE begins to provide directions on how to specify properties often through tags. The use of these tags is defined in the MARTE meta-model. This being said, there does not appear to be automated analysis tool support at this time.

## **E. Extension Mechanisms**

AADL, being developed under the paradigm of strict semantics, is more limited than UML in its mechanisms to extend the language. However, that does not mean that extension points in AADL do not exist. AADL's main extension mechanisms are annex libraries and custom property sets. An annex library extends the language to enable the modeling of entities not represented by the core language, one notable example of which is the AADL Behavior Annex. As previously stated, AADL does not provide a mechanism to detail state-machine like behavior by default; the Behavior Annex embeds textual notation of state-machine behavior in the AADL textual model. Custom property sets enable the modeler to annotate components and connectors with domain-specific information. For example, avionics components could be annotated with a weight property if the modeler was interested in the total weight of the avionics plagram. Neither annex libraries nor custom property set alter (by changing semantics or adding components) the core AADL language; these mechanisms only extend the core language providing domain specific notation. New automated analysis plug-ins that utilize these extensions can be easily added to OSATE, because it is based on Eclipse,

The basic extension mechanism in UML is the stereotype, which brings components into a specific domain and is denoted by encasing the stereotype identifier in <<>>. For example, generic UML does not provide the explicit semantics to model individuals in an organization such as a secretary, manager, and programmer, but the modeler can create a stereotype, <<Manger>>, for example, and use this stereotype in diagrams. Meta-models are another major extension point of UML and SysML. Meta-models can be used to define the organization and relation of domain specific entities in a diagram. Combining meta-models and stereotypes, an entire profile can be created to address the concerns of a particular domain. For example, MARTE is built on a defined meta-model in order to address the concerns of a real-time embedded system.

## **F. Model Comparison and Representation of Time**

It is also important to express temporal relations of concurrently executing software entities. The modeling language must be able to satisfactorily support this aspect of real-time embedded software systems especially in a distributed computing system.

UML provides the means to model sequential actions in a software system via sequence diagrams, activity diagrams, and state-machine diagrams. It is possible to denote some timing values on these diagrams in order to detail execution time, periodicity, or some other basic temporal relation.

Figure 3 shows three objects in a system. One handles data acquisition, one data processing, and another generates commands. The DataProcessing object is annotated to have an execution time of less than 2 milliseconds. Without having to understand much about the operation of the system, one can ascertain the intent of the description - the DataProcessing objects completes its execution in under 2 milliseconds. This attribute is not available in UML - it must be created and added by the modeler. The constructs to describe jitter, resolution, and frequency of execution are also not available in UML and they too have to be added by the modeler. Even with this annotation, the

sequence diagram and other behavioral diagrams only describe the sequential interaction of software entities and do not explicitly express any concurrency between the entities.

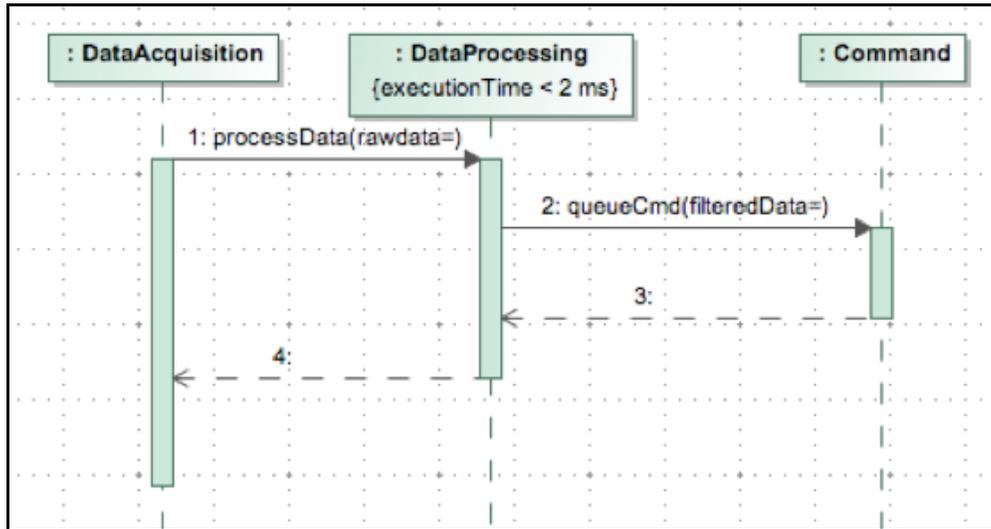


Figure Error! Unknown switch argument. - System Sequence Diagram

AADL was built, in part, to describe exactly that: the concurrent nature of executing software components. The AADL thread component is central to this end. The real-time characteristics expressed through AADL properties, and the notion of time for that matter, serve mostly to describe how-not when-a concurrent thread executes, especially in relation to threads executing on other processors. In Figure 3, the DataProcessing object is annotated to describe the execution time. In AADL the Compute\_Execution\_Time property, applied to an AADL thread component, explicitly describes execution time in relation to other threads in the AADL model. In a similar manner, properties such as period, priority, and execution protocol are defined using built-in AADL properties.

Figure 4 shows a basic AADL textual model containing three concurrently executing threads (similar to the objects depicted in Figure 3), and several properties common to describing real-time systems. The concurrent nature of these threads is explicitly expressed by the semantics of the AADL thread component. The responsibility for executing these threads is hidden by the AADL processor component (omitted from the model in Figure 4). The AADL processor component serves as the abstraction for the real-time scheduler as well as the real-time operating system. A large number of AADL processor components can be used to handle any number of AADL thread components in an attempt to model a distributed system.

```

process basicProcess
end basicProcess;

process implementation basicProcess.imp
  subcomponents
    DataAcquisition: thread basicThread.da;
    DataProcessing: thread basicThread.dp;
    Command: thread basicThread.cmd;
  end basicProcess.imp;

thread basicThread
end basicThread;

thread implementation basicThread.da
  properties
    Period => 50 Ms;
    Compute_Execution_Time => 1 Ms .. 1 Ms;
    Dispatch_Protocol => Periodic;
    SEI::Priority => 5;
  end basicThread.da;

thread implementation basicThread.dp
  properties
    Period => 20 Ms;
    Compute_Execution_Time => 2 Ms .. 2 Ms;
    Dispatch_Protocol => Periodic;
    SEI::Priority => 1;
  end basicThread.dp;

thread implementation basicThread.cmd
  properties
    Period => 20 Ms;
    Compute_Execution_Time => 1 Ms .. 1 Ms;
    Dispatch_Protocol => Periodic;
    SEI::Priority =>5;
  end basicThread.cmd;

```

**Figure Error! Unknown switch argument. - AADL Textual Model**

The underlying semantics of AADL were not designed to relate the execution of concurrent threads to real-world time (physical time), which is acceptable in instances where physical time is not important. However, in some applications, where events must occur at a specific instant in physical time, the ability to relate timing in the software system to the environment is paramount. The ability to model time at different levels of abstraction is a strength of MARTE.

Figure 5 shows the software entities of this example using MARTE. In the “GenericAPI” package, “myPartition” class is stereotyped with the <<MemoryPartition>> tag. This is similar to an AADL process component in that it represents a partitioned software space. The “BasicThread” component is stereotyped by the <<SwScheduableResource>>, which maps to the AADL thread component. Here, hardware is also modeled, specifically two processors. Through the MARTE concept of “allocation” software entities are mapped to hardware elements. However, not rendered in Figure 5, consider “daThread” and “procThread” are allocated to processor “Proc1” and “cmdThread” is allocated to “Proc2”. From a static view, there is nothing incorrect with this organization of the architecture. However, from a dynamic perspective, there is little concept to how the software is temporally related. Another diagram is needed to illustrate these aspects of this simplistic distributed system.

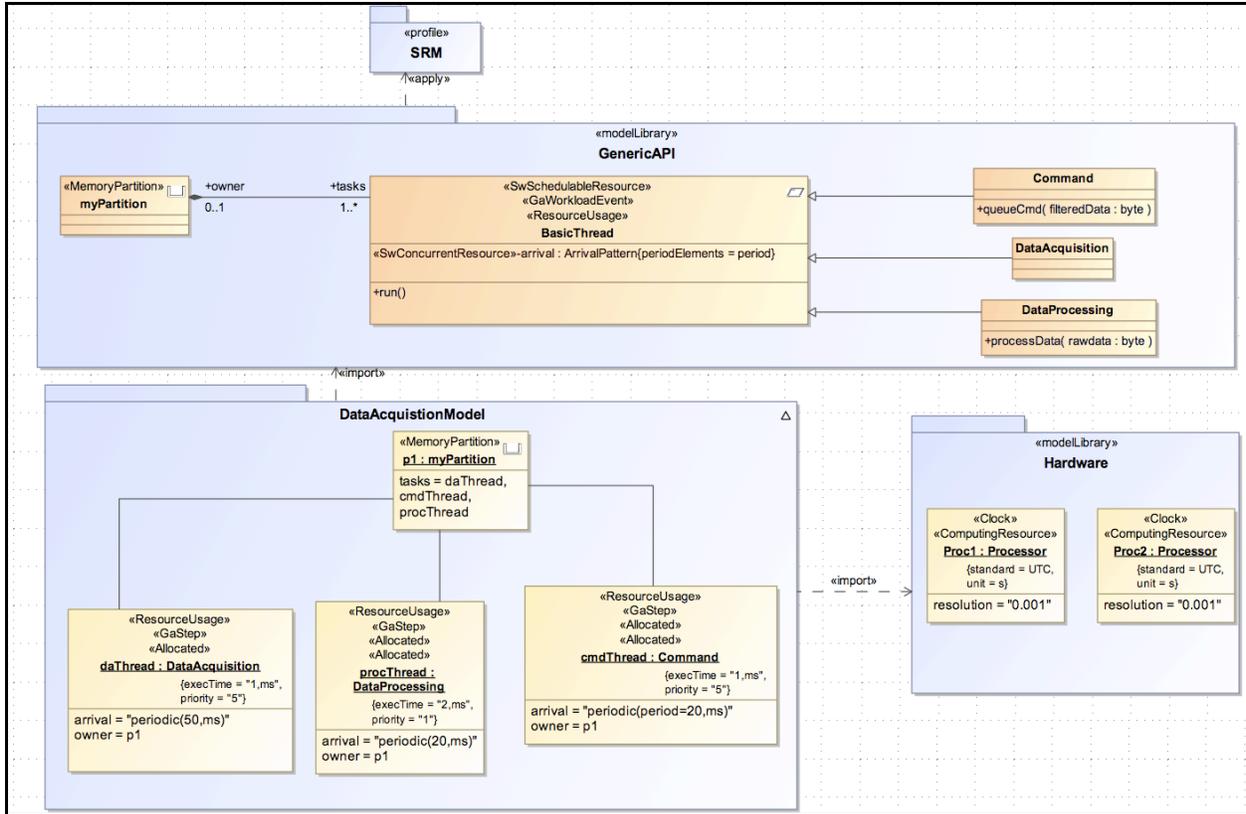


Figure Error! Unknown switch argument. - MARTE Thread Example

Figure 6 uses components from the Time library in the MARTE profile to assist in relating the two processors. Each processor has an independent notion of time, starting at zero and increasing, referred to as chronometric time. This time base is composed of a discrete set of instants. These two processors are related in time by a mission clock represented by the “MissionClock” class. The time nature of this clock is “dense” indicating a continuous time base as opposed to a discrete time base. Using a <<clockConstraint>> stereotype, the modeler can annotate the model explicitly specifying the time relation between each processor. Clock constraints are specified by the Clock Constraint Specification Language (CCSL) found in the MARTE specification<sup>1</sup>. This figure is adapted from a model depicted in the Time chapter of the MARTE profile.

The constraint present in the <<clockConstraint>> introduces a discrete, conceptual clock “c”, that is based on the mission clock with each discrete instant occurring every 1 ms. The next two lines indicate that each countable period on the two processors occurs every 10 instants on clock c. The last line indicates that “Proc2” is offset from “proc1” by no more than 1 ms.

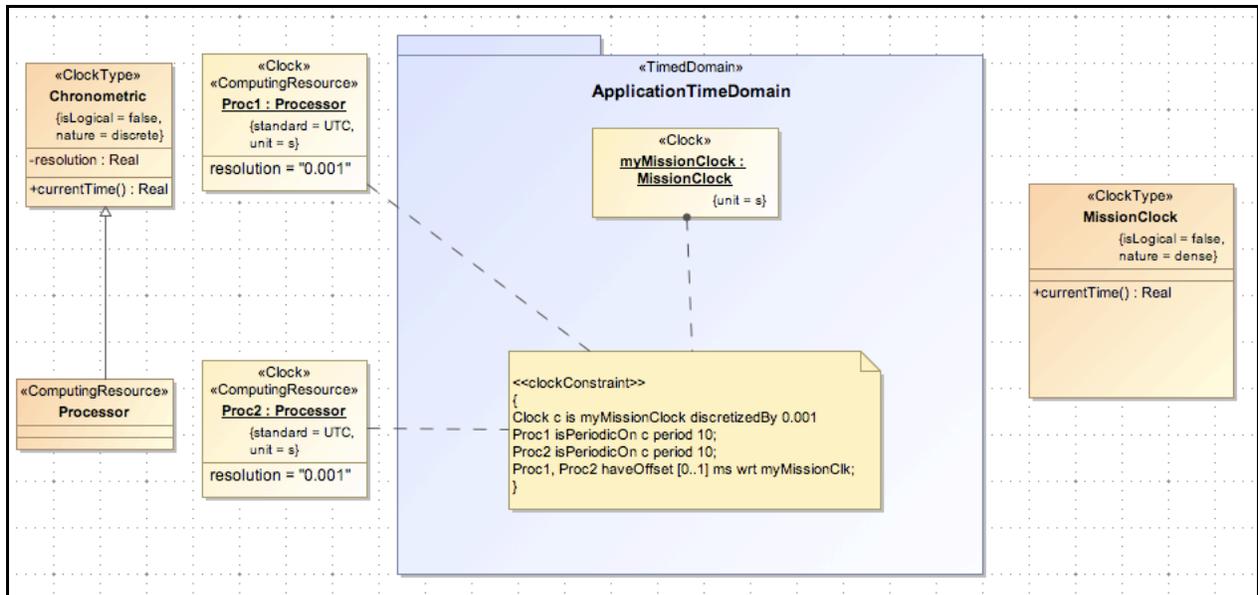


Figure Error! Unknown switch argument. - Time Model

In execution, this relationship would be enforced by either a software signal or real-time interrupt to synchronize the concurrently executing threads. MARTE provides the constructs to detail “when” the threads execute, in addition to “how”.

MARTE’s mechanisms for the expression of time are far more extensive than what is illustrated in the above example. For a deeper understanding, it is recommended that the reader refer to the MARTE Specification on provided on the OMG website (<http://www.omgmarTE.org>).

## G. Limitations

Some limitations of these languages have been indirectly discussed in previous sections. Here, the key limitations are presented.

AADL’s strength is its ability to model multi-threaded systems, or systems with concurrent resources.. Consequently, AADL would not be appropriate for modeling a system with a single thread of execution. Nor would AADL be useful in modeling the components of a graphical user interface (GUI), because such components are behavior-centric and AADL cannot model behavior-centric software systems. If a real-time system had a graphical user interface, a possible solution might be to model the real-time portion in AADL, and the GUI portion in UML. AADL cannot be used to capture use-cases or any other function of software architecture other than the execution of software components and how they relate to avionics hardware.

AADL abstracts the functions of the RTOS such as the scheduler, watchdog timers, and signals that would originate from the OS. A solution here might be to model such functions as AADL Devices, although this is not the most explicit representation of these real-time concepts. AADL also does not explicitly model the clock and timer concepts found in MARTE.

UML is well suited for the architecture and design of object-oriented and/or business oriented systems and can readily be adapted to a stakeholder’s domain. The ease with which diagrams can be customized to answer the concerns of a stakeholder can also be considered a pitfall. Furthermore, it is difficult with a large number of diagrams, generated either by design or accident, to maintain a consistent, semantically-correct specification. Tools can help mitigate this issue by indicating if relationship rules are being violated. Because MARTE is an extension to UML, there is a possibility of a large number of diagrams being generated with that language as well. Furthermore, MARTE has a very complex meta-model. Finally, the UML Profile for AADL lacks tool support. This is not surprising, since MARTE has been identified as the path forward for a UML based real-time modeling language.

#### **IV. Conclusions**

Table 1 presents a summary of the language comparisons with respect to the dimensions of comparison presented in this paper. In assessing modeling strategies for embedded real-time software systems, the selection is between AADL, UML, SysML and MARTE. As previously states, the UML Profile for AADL is being submitted as a part of the MARTE profile. Due to the limited support and development of the UML Profile for AADL, it has been omitted from this table.

**Table Error! Unknown switch argument. - Comparison Summary**

	<b>AADL</b>	<b>UML</b>	<b>SysML</b>	<b>MARTE</b>
<b>Scope</b>	<ul style="list-style-type: none"> <li>Embedded real-time software and avionics hardware</li> </ul>	<ul style="list-style-type: none"> <li>Object-oriented software solutions</li> <li>Business software solutions</li> </ul>	<ul style="list-style-type: none"> <li>Systems engineering</li> <li>Includes requirements and parametric (equation) diagrams</li> </ul>	<ul style="list-style-type: none"> <li>Embedded real-time software and avionics hardware</li> </ul>
<b>Formalism</b>	<ul style="list-style-type: none"> <li>Strict component connector semantics</li> <li>Entity relationships enforced by language specification</li> </ul>	<ul style="list-style-type: none"> <li>Diagram-centric</li> <li>Underlying formalisms defined by UML meta-model</li> </ul>	<ul style="list-style-type: none"> <li>Diagram-centric</li> <li>Underlying formalisms defined by SysML meta-model</li> </ul>	<ul style="list-style-type: none"> <li>Diagram-centric</li> <li>Underlying formalisms defined by MARTE meta-model</li> <li>UML Profile for AADL</li> <li>Guidelines for modeling ant each level of abstraction</li> </ul>
<b>Architectural Coverage</b>	<ul style="list-style-type: none"> <li>Execution software entities</li> <li>Real-time hardware components</li> <li>Static representation of software</li> </ul>	<ul style="list-style-type: none"> <li>Implementation, execution, and compile time entities.</li> <li>Static and behavioral representation of software</li> <li>High level architecture to detail design</li> </ul>	<ul style="list-style-type: none"> <li>Conceptual, systems view</li> <li>Static and behavioral representation of system</li> </ul>	<ul style="list-style-type: none"> <li>Implementation, execution, and compile time software entities.</li> <li>Real-time hardware components</li> <li>High level architecture to detail design</li> </ul>
<b>Tool Support</b>	<ul style="list-style-type: none"> <li>Open Source AADL Tool Environment (OSATE)</li> <li>Ocarina</li> </ul>	<ul style="list-style-type: none"> <li>MagicDraw</li> <li>Visual Paradigm</li> <li>Papyrus UML</li> </ul>	<ul style="list-style-type: none"> <li>MagicDraw</li> <li>Visual Paradigm</li> <li>Papyrus UML</li> </ul>	<ul style="list-style-type: none"> <li>MagicDraw</li> <li>Visual Paradigm</li> <li>Papyrus UML</li> </ul>
<b>Extension Mechanisms</b>	<ul style="list-style-type: none"> <li>Custom property sets</li> <li>Annex Libraries</li> <li>Custom plug-ins for OSATE</li> </ul>	<ul style="list-style-type: none"> <li>Stereotypes</li> <li>Meta-models</li> <li>Profiles</li> </ul>	<ul style="list-style-type: none"> <li>Stereotypes</li> <li>Meta-models</li> </ul>	<ul style="list-style-type: none"> <li>Stereotypes</li> <li>Meta-models</li> </ul>
<b>Limitations</b>	<ul style="list-style-type: none"> <li>Multi-threaded systems</li> <li>Abstraction of real-time operating system components</li> <li>No behavior modeling</li> </ul>	<ul style="list-style-type: none"> <li>No strict formalism</li> <li>Possibility of large number of diagrams</li> </ul>	<ul style="list-style-type: none"> <li>Relegated to systems engineering domain</li> </ul>	<ul style="list-style-type: none"> <li>Possibility of large number of diagrams</li> <li>Complex underlying meta-model</li> </ul>

AADL and MARTE provide the richest set of modeling tools for embedded, real-time systems. UML and SysML are not designed to model in the real-time systems domain. The UML profile for AADL is being adopted as a part of the MARTE profile. The strategy for combining the two provides a bridge to translate AADL models in to MARTE, or vice-versa, should the need arise during a modeling effort.

MARTE has an expansive array of packages and components to address the schedulability, performance, and timing concerns of a real-time system. With UML at its core, MARTE can be extended to model and answer domain specific concerns. Coupled with UML and SysML, MARTE provides almost complete architectural coverage, from a systems engineering level of abstraction, down to the detailed design. MARTE can be used to model the compile time, execution, and deployment nature of the software. Tool support for MARTE is growing as the specification is maturing. There is still a need for a tool to conduct automated analyses on MARTE models in order to assess the fitness of a software design in relation to its software requirements.

Even with a small library of components the Architectural Analysis and Design Language continues to be a suitable choice in designing real-time systems, due in part to its strict semantics. OSATE provides a firm foundation for developing models in AADL and performing automated analyses when using AADL properties. Since OSATE is built on the Eclipse RCP, the library of automated analyses can be extended by developing plug-ins as needed. While AADL is not intended to model the compile time or deployment architectures of real-time software, AADL can clearly specify the execution/run-time nature of the software and avionics hardware components.

In a complex software project, especially one involving real-time systems, no single modeling technique is the answer. The needs of the stakeholders can be a driving factor, as well as the expertise of the software architect. For a complete architecture, the software architect will likely need to utilize elements from SysML, UML, MARTE, and AADL. For example, use cases can be captured in UML and requirements would be captured in SysML. During architecture and design, the high level architecture would likely be modeled using SysML in order to convey systems concepts to a Systems Engineer. The compile time and deployment architecture could be modeled in MARTE, while the execution nature of the software modeled in AADL. The relationship between MARTE and AADL is strong enough to make this a possibility.

As real-time, embedded software systems continue to grow in applicability, there is a need to support the correct design and documentation of these systems. Fortunately, the tools and techniques to accomplish these tasks are rapidly maturing. The limitations of the modeling techniques presented are mitigated by the ability to augment models generated in one language by diagrams specified in another.

## **V. Acknowledgements**

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. This research was funded by the NASA OSMA Software Assurance Research Program.

## References

1. Vestal, Steve, Larry Stickler, Dennis Foo Kune, Pam Binns, and Nitin Lamba. *AADL - Documents*. Honeywell, 16 June 2004. Web. <[www.aadl.info/aadl/documents/AADL-MetaH%20for%20LAS.pdf](http://www.aadl.info/aadl/documents/AADL-MetaH%20for%20LAS.pdf)>.
2. A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems, Beta 2. June 2008.
3. OMG Systems Modeling Language v1.1. <http://www.omg.org/spec/SysML/1.1>. Specification. November 2008.
4. Booch, Grady, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide*. Reading Mass: Addison-Wesley, 1999. Print.
5. Feiler, Peter H., David P. Gluch, and John J. Hudak. "The Architecture Analysis & Design Language (AADL): An Introduction." CMU/SEI-2006-TN-011. 2006.
6. Introduction to OMG UML. [http://www.omg.org/gettingstarted/what\\_is\\_uml.htm](http://www.omg.org/gettingstarted/what_is_uml.htm). October 2009.
7. The Official OMG MARTE Web Site. <http://www.omgmarte.org/>. October 2009.
8. Orcarnia. TELECOM ParisTech AADL corner. <http://penelope.enst.fr/aadl/>. October 2009.
9. OSATE: Open Source AADL Tool Environment. <http://www.aadl.info>. October 2009.
10. Papyrus UML. <http://www.papyrusuml.org>. November 2009.