Diagrams and Languages for Model-Based Software Engineering of Embedded Systems: UML and AADL

Dionisio de Niz
dionisio@sei.cmu.edu

Even after years of research and practice in computer science and, in particular, in software engineering, software projects are still largely risky and unpredictable. There is significant evidence to support this observation. Consider, for instance, a NIST (National Institute of Standards and Technology) study in 2002 that found software errors cost the U.S. economy $59.5 billion annually, about 0.6 percent of the national gross domestic product [NIST 2002].

Based on that total, software users and developers pay more because of error-ridden software than gamblers do at the slot-machines and tables in Las Vegas, Atlantic City, and all other commercial venues that provide gaming. Gamblers accept the risk associated with the roll of a dice; software users should not have to.

It is not that developers do not apply resources to discovering and fixing errors. They do. The same NIST study reported that nearly 80% of the money spent in development goes to correcting defects. Yet, software, unlike almost any other product, is provided to customers with a high-level of errors [NIST 2002].

One recent study to uncover the causes of software project risk was performed by the Committee on Certifiably Dependable Software Systems of the National Academy of Sciences (NAS) [NAS 2007]. Two of their key observations have a strong impact on the purpose of model-based software engineering:

1. In software development, there is no substitute for simplicity. While difficult to achieve, simplicity is worth the cost. One way to achieve simplicity is to develop high-level software structures that limit the complexity of interactions among components. Such structures are known as software architecture.
2. The behavior of the software goes beyond the software itself to involve the environment with which it interacts. This environment includes hardware and the physical world. Hence, any property of the software is, in part, defined by assumptions made about it by the environment. Furthermore, any claim on the software needs to be explicit and unambiguous and captured in the proper form to enable automated analysis.

It is worth noting that the need for the automated analysis of claims has a twofold benefit. On the one hand, it enables the designer to cope with a level of complexity that humans cannot handle (but machines can). And on the other hand, automated analysis removes human interpretation from the verification of claims—and in so doing eliminates the possibility of ambiguity.

MBE tools support the automated analysis of software system architecture

Model-based engineering (MBE) tools for software engineering recognize the importance of architecture and automated analysis. The tools we compare in this discussion, the Unified Modeling Language (UML) and the Architecture Analysis and Design Language (AADL) facilitate the modeling of software architecture and provide elements to understand it.

BASIC COMPARISON

UML provides a set of diagrams to depict software structures graphically. These diagrams appeal to practitioners and help them tackle complex software structures. However, while its individual diagrams are useful to depict software structures, UML cannot fully define the relationships between diagrams. The diagrams are developed as separate entities that express different aspects of the software, not as parts of a common construct. As a result, the consistency across diagrams is largely left to be resolved by the designer. Notwithstanding that issue, UML has been broadly adopted due to the way it reflects the concerns and communications needs of programmers and software designers.

AADL comes from a computer language tradition, rather than a diagrams tradition. AADL, like its predecessor MetaH, produces language-based modeling artifacts. AADL was developed as a programming language

not only to define the textual representation of software architecture but also (and more importantly) to formally define the syntax and semantics. (In addition to textual representation, AADL allows the software designer to depict the system graphically.)

System designers can use UML to diagram functional structures, and AADL to define runtime behavior

As a result, descriptions in AADL comply with the syntax and semantics of the language and can be verified by the syntactic and semantic analyzer of the language to ensure that the description is analyzable and consistent. In other words, constructs in a model are checked by the compiler to verify that they are "legal" (e.g., a thread cannot contain processes). They are also assessed for correctness (e.g., defining a periodic thread that does not have specified period). Verification of the description happens in the same fashion a compiler checks that a program is properly structured, consistent, and semantically correct to be able to produce executable code.

Any software description in AADL is analyzable and has an unambiguous interpretation (as a program would have for a compiler). Analyses are built on top of the language constructs, further the emphasis on unambiguous interpretation.

A summary of the basic comparison between AADL and UML can be seen in Table 1.

Table 1. Basic Comparison of UML and AADL

	UML	AADL
Origin	Diagrams tradition	Language tradition
Purpose	Depict functional structures	Define runtime behavior
Representation	Diagrams; graphic	Textual and graphic
Verification	---	Automated analysis
Current Domains of Use	Software, business processes, and many others	Embedded and real-time software system

HOW UML AND AADL ACCOMMODATE EXTENSIONS

Both UML and AADL provide extensions to accommodate new constructs for the modeling artifacts.

UML Extension Mechanisms

UML has three extension mechanisms:

1. stereotypes
2. tagged values
3. constraints

These mechanisms are typically bundled together in a profile that represents a modeling dialect for a specific purpose.

Stereotypes are new model elements derived from core ones. These stereotypes can later be applied to UML model elements to identify them as these special elements. Stereotypes can have special attributes. In addition, stereotypes can be associated with a special graphical element.

Tagged values are properties that associate keyword-value pairs to model elements. They are used to extend the description of elements with annotations for a specific purpose of the profile.

Finally, constraints are restrictions that are expressed in a special language called object constraint language (OCL). These restrictions allow the specification of semantic restriction for the construction of models, taking the role of what a syntactic and semantic analysis does in a compiler (e.g., restricting an invoice to be associated with only one client). The consistency of the rule in OCL is in the hands of the designer of the profile.

AADL Extension Mechanisms

AADL provides an extension construct called annex to add complementary description elements for different kinds of analysis not covered with the core elements. These annexes are embedded in descriptions of the core language and can make references to constructs in it. Annexes are language extensions, which means that, along with the annex, a compiler is built to analyze annex submodels for syntactic and semantic integrity. The defining of annexes is standardized to assure completeness and consistency.

AADL analysis tools, such as the Open Source AADL Tool Environment (OSATE), implement annexes as parsers, name resolvers, and semantic checkers. They extend the basic checking of the core language and are used in a cascading integrated compilation process to provide full consistency verification.

Along with the annexes, the AADL defines property set extensions. In a way, property set extensions in AADL are similar to the tagged values of UML. However, because they live in the language, property set extensions offer the possibility to refer to other language constructs, define their types (e.g., real, integer, range of integers), or extend other properties (e.g., the Period of a thread extends the Time property type).

The comparison of the extension mechanisms is summarized in Table 2..

Table 2. Summary of Extension Mechanisms

UML or AADL	Type	Purpose
UML	Stereotype	New model elements derived from core ones
	Tagged Value	Properties that associate a key-word-value pair to an element or more than one element
	Constraint	Semantic restriction for model construction
AADL	Standard annex	Language extension
	Property set extension	Project-specific construct

WHAT UML AND AADL FOCUS ON

UML was conceived as a way to model functional structures of software; AADL, to model and analyze runtime architecture. As both have been more widely adopted, they have been used in areas that complement their core areas of use.

UML Core Focus

UML focuses on three aspects of the functional structures: data, interaction, and evolution.

- The data is modeled in **class** diagrams. Classes are central pieces of data modeling in UML.
- Interaction is modeled with a **sequence** diagram or a **collaboration** diagram. Thee diagrams describe how classes interact to achieve a specific task of the application.
- **Evolution** in this context defines the modeling that explicitly describes states of the systems and their transitions. Evolution is typically modeled with **state** diagrams embedded in objects.

While UML offers other diagrams, classes, sequence, and state diagrams strongly define the main functional structure of the software.

SysML and the MARTE profile are extending the use of UML diagrams into embedded and real-time system design

UML Extended Areas of Use

Recently, the UML community has been working on enabling multiple analyses to prove properties of the modeled system. Two of these efforts related to embedded systems are **SysML** and the **MARTE** (Modeling and Analysis of Real-Time and Embedded systems) profile. SysML allows the capturing of the interactions with the physical world in a mathematical model and the verification of properties on it. MARTE is intended to add modeling capabilities to verify real-time properties such as timeliness and schedulability.

SysML provides two fundamentally new diagrams:

1. requirements diagram
2. parametric diagram

These extensions modify the core diagrams instead of using the extension mechanisms (i.e., stereotypes, tagged values, and constraints). The requirements diagram supports a stronger focus on requirements and traceability. The parametric diagram expresses the relationship between the software and the environment.

Using the parametric diagram, a designer can model the software-environment relationship mathematically to verify, for instance, whether the software can control the environment or other properties. However, the modeling with parametric diagrams is focused on continuous time where computations are instantaneous. This focus turns to a disadvantage when the ideal model is translated into real executable software, where non-instantaneous execution jeopardizes the validity of the analyses.

The purpose of the **MARTE profile** is to enable the analysis of real-time properties using the rate-monotonic theory and code generation in the presence of different operating systems. In MARTE, multiple stereotypes are defined. The new stereotypes specify elements to model three aspects:

1. software resource model
2. hardware resource model
3. the allocation of the software model to the hardware model

This resource modeling is based on the rate-monotonic theory and includes a mapping between a generic OS API[1] and specific OS APIs to be able to generate code automatically. Using MARTE, a designer models the system with multiple functional, runtime, and hardware diagrams. Then, connections between the diagrams are used to model the allocation of entities from one diagram to another. However, the consistency between these diagrams is left to the designer.

1. Operating System Application Programming Interface

The MARTE profile incorporates experience from the AADL community with respect to modeling the runtime and hardware architectures. Furthermore, some members of the AADL standard committee are on the MARTE committee.

Focused on the unambiguous specification and analysis of runtime architecture, the AADL has been extended through error model and behavioral annexes

AADL Core Focus

The core focus of AADL is runtime architecture modeling and analysis. Runtime architecture is the software structure that defines the final execution sequence of instructions. This structure is defined by threads, processes, processors, and their interactions (data, event, and event data communication) that encapsulate the functional modules that they execute. Runtime architecture provides the software system with specific quality attributes such as timeliness, fault-tolerance, or security.

AADL language semantics, enforced by compilation techniques, provide a clear execution semantics that is defined as a **hybrid automaton** in the standard document. A hybrid automaton is a mathematical model for describing how software and physical processes interact. The AADL hybrid automaton defines, unambiguously, the specific combinations of events that trigger or stop the execution of the different elements of the model. These combinations can be due to interactions or the passage of time.

This execution model in AADL encodes the most effective structures used by embedded systems developers and assumed by the theory of real-time systems. For instance, it encodes periodic and aperiodic threads, periodic data sampling, state variable communications, event-based transfer of control, and isolation strategies for memory and time such as the ones found in partition architectures in the style of ARINC 653 [ARINC 653 2003].

AADL Extended Areas of Use

Two AADL annexes (extensions) have expanded the capabilities of AADL: the error annex and the behavioral annex. The error annex enables the detailed, state machine description of potential errors in the architecture, on which a designer can create theoretically strong models such as Markov claims. The behavioral annex permits the description of functional behavior to enable formal verification in the style of model checking.

The error and the behavioral annexes are now standard annexes to the language. However, the flexibility of the annex mechanism allows the designer to add precise extensions based on need. For example, tool developers at the Carnegie Mellon[1]® Software Engineering Institute (SEI) have developed multiple experimental analyses for AADL models that include project-specific annexes for different domains such as security and fault propagation.

1. Carnegie Mellon is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

The summary of the comparison of the areas of use for UML and AADL is presented in Table 3.

Table 3. Focus Areas

UML or AADL	Core Area	Extended Areas
UML	Functional structures	Analysis through SysML and MARTE
AADL	Runtime architecture modeling and analysis	Error handling through error model annex; formal verification through behavioral annex

HOW UML AND AADL ARE COMPLEMENTARY

In their core purposes, UML and AADL are complementary. UML focuses on the functional structures of software abstracted from the runtime architecture. AADL, on the other hand, focuses on the runtime architecture, while the functional structure is extracted away.

As a result, a system designer can exploit the strengths of both, using them in complementary roles. The SEI, in collaboration with Kennedy-Carter Ltd., has developed a mapping between xUML and AADL [ICECCS 2007]. (xUML is a form of UML with executable semantics.) In this mapping, xUML is used to model the functional structures in what is called the Platform Independent Model (PIM), and AADL is used to describe the runtime architecture in what is called a Platform Specific Model (PSM).

The PIM model which defines how data is transformed through the system, does not specify the timing of such transformations, the effect of distributing these transformation to multiple processors, or how the failure of one processor can affect them. For this purpose, AADL is used in the PSM model. In this model, the semantics of AADL allows the designer to clearly model when transformations happen in parallel (e.g., the antilock brake system [ABS] module monitors the locking of wheels while the fuel module controls the mixture of fuel and air in the engine), when such transformations need to be redundant (e.g., having two computers to monitor the brakes, so if one dies, the other can continue to work seamlessly), or when they need to be isolated from other applications (e.g., the DVD system and the ABS should be in different computers, so that a bug in one would not stop the other).

UML AND AADL: TOGETHER FOR HIGHER PRODUCTIVITY

Though they differ significantly, UML and AADL can be used together effectively to improve the predictability of real-time and embedded systems

UML diagrams, with their communication strength, used in conjunction with AADL modeling and analysis, with their precision for runtime architecture, create a powerful combination to improve predictability in the development of embedded and real-time systems. In particular, they can lessen risk in crucial aspects of embedded and real-time systems through the

- physical modeling capabilities (and potential for automated analysis) of SysML
- functional modeling of UML
- runtime architectural modeling of AADL (and its analysis capability) and MARTE
- error annex of AADL

In turn, lowered risk can turn into savings for individuals and organizations. The NIST study that pegged the annual cost of software errors at $59.5 billion also asserted that an estimated $22.2 billion of that amount could be saved with improved "testing infrastructure that enables earlier and more effective identification and removal of software defects" [NIST 2002]. Modeling and analysis, using UML and AADL, can provide designers with crucial insight into software structure and behavior.

REFERENCES

[ARINC 653 2003]
Lynuxworks. *ARINC 653 (ARINC 653-1).* http://www.lynuxworks.com/solutions/milaero/arinc-653.php (2003).

[ICECCS 2007]
Feiler, Peter H., de Niz, Dionisio, Raistrick, Chris, & Lewis, Bruce A. "From PIMs to PSMs," 365–370. 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007). Auckland, New Zealand, July 2007. http://doi.ieeecomputersociety.org/10.1109/ICECCS.2007.25

[NAS 2007]
Jackson, Daniel, Thomas, Martyn, & Millett, Lynette I., Eds. Software for Dependable Systems: Sufficient Evidence? Washington, D.C.: National Academies Press, 2007. http://books.nap.edu/catalog.php?record_id=11923

[NIST 2002]
National Institute of Standards and Testing. *Planning Report 02-3: The Economic Impacts of Inadequate Infrastructure for Software Testing (May 2002))* (RTI Project Number 7007.011). Research Triangle Park, NC: RTI, 2002. http://www.nist/gov/public_affairs/releases/n02-10.htm