

AADL and Simulink¹

Peter Feiler
SEI
Sept 27, 2006
phf@sei.cmu.edu

Introduction

Simulink models are created to represent the control application as well as the plant being controlled. These models may represent a single control application, e.g., an electronic throttle control, or a collection of control applications that may interact with each other, e.g., cruise control and braking system. Each control application and controlled plant element may be modeled in more than one level of fidelity.

When running simulations all control application elements and controlled plant elements are simulated by a simulation engine that determines the execution order of the individual blocks. In other words, there is no concurrency in the execution.

The execution models can be translated into executable code through the Real-time Workshop tool. Through this tool a model is translated into a single executable program. A controller may be translated into a single program, or the controller together with the simulation model of the plant it control may become an executable program. The primary role of AADL is to specify the run-time architecture of the embedded application as a set of concurrent communicating tasks that execute on one or more processors, either interfacing to the real physical plant or to a simulation of the physical plant.

In the following sections we will exercise several use scenarios to illustrate the use of AADL in the context of Simulink models.

Documenting the Plant Interface

In this scenario we assume that the user is developing a Simulink model of a single controller. In doing so the user defines an interface to the plant being controlled and creates a model of the plant as well.

We can use AADL to define the controller as an AADL system component and the plant as an AADL device component. A reference to the Simulink model file is recorded as a Source_Text property with each of the components.

For both components the user defines ports and specifies assumptions being made about that characteristics of the data. Matlab/Simulink typically allows the user to specify the base type of data representations, i.e., signed/unsigned 8/16/32 bit integer, etc., and will flag any mismatch in base type representation. The data dictionary associated with the state flow allows users to also define application data types.

¹ Copyright Carnegie Mellon 2006

The AADL specification of the control component allows the user to specify characteristics of the data being passed, i.e., both the base type (e.g., uint16) and the application data type (e.g., speed). Furthermore, the user can specify the measurement unit of the data being passed (e.g., kph or mph), the range of values. The user can also specify characteristics of the data stream, such as maximum delta between successive values (a possible characteristic of a sequence of set-points), or the rate of missing elements in the data stream (e.g., the rate at which a sensor may not deliver a reading, or a processing element of the data stream may miss a deadline or otherwise drop a data element). [SVM 2004, Krogh 2003, Krogh 2004, Emmeskay 2006].

Annotations on components and ports can also be used to document assumptions being made about other components, i.e., information that is not explicitly documented. For example, a controller may make assumptions regarding the lag in the response of a physical plant to a control command. This capability has been demonstrated in the context of AADL with the Assumption Management Framework (AMF) work at University of Illinois Champaign-Urbana [Tirumala 2006].

Documenting the Interface between Simulation Blocks

In this scenario we would like to take advantage of the capability in AADL to more precisely specify the interface between components (blocks) within a control model. Ideally we would like to do this within Simulink.

This can be done by extracting architectural information from the Simulink model and annotating it. In that context one issue to be addressed is that of changes to the Simulink model and the corresponding architecture model. This can be addressed in one of two ways.

The first way of keeping a Simulink model and the corresponding architecture model synchronized was demonstrated in the System Verification Manager toolset [SVM 2004]. In this case the architecture model is extracted from the Simulink model to a level of detail that is controlled by the user. The architecture model is then annotated. If changes are made to the Simulink model the architecture extraction method is able to identify changes between the existing architecture model and the newly extracted one, and update the existing one accordingly without losing annotations to components that exist in both models.

This can be taken one step further to the second way. In this case, once the architecture model is established, the Simulink model is split into appropriate pieces, one for each leaf node in the architecture model. The composition of these Simulink model pieces is already recorded as part of the architecture model. The user can now manipulate the architecture in the architecture model and modify the algorithmic aspect of the model through one of the model pieces. The composite Simulink model is then generated from the pieces and the architecture.

In summary, we are modeling details within an component that itself may execute concurrently. This is not the major focus of AADL. However, it does provide support for doing so and can be used to complement model consistency checking that may be lacking in the component domain modeling language.

Models of Multiple Fidelity

Typically multiple Simulink models of different fidelity are created for a physical component and variants are created for control algorithms. These collections of models represent the same system, thus, have a common conceptual system architecture.

In AADL components have a component type specification that represents the external interface of components. Multiple component implementations can be defined for the same component type. These implementations may vary solely in their reference to the Simulink model file (the value of the `Source_Text` property), or implementations may vary in their internal structure, i.e., the subcomponents and their connectivity, where the subcomponents themselves may have references to Simulink model files or blocks within Simulink model files. A more detailed discussion of how to model system variants and system configurations can be found in [Feiler 2006].

The System Verification Manager (SVM 2004) has also addressed this issue and demonstrated how multiple model variants can be associated with the same architecture model, how a common architecture model can be derived from architecture model extractions of individual Simulink models. SVM's focus then was on automating the verification of these models by automating the process of running validations (in form of simulation runs or model checking) in order to demonstrate that requirements are met. This work is now being commercialized by Emmeskay [Emmeskay 2006].

Concurrent Units of Execution

At some point in the development cycle a project will be translating a conceptual system architecture that has been modeled in its detail in Simulink, into an embedded software system. We now want to be able to describe the runtime architecture of this embedded software system. This is the main focus of the AADL.

Using AADL the user can now define a runtime architecture by identifying the concurrent units of execution (AADL threads), their rate of execution, deadline, worst-case execution time. Similarly, the user can specify the interactions between those concurrent tasks and characterize the timing semantics of the port connections of sampled data ports in terms of mid-frame (immediate) and phase-delayed (delayed) communication.

The user identifies the executable code of the thread either by referring to the code generated from a Simulink model through Real-time Workshop, or by referring to the portion of a Simulink model that is passed to Real-time Workshop to generate the code. In other words, the AADL model is annotated with information that allows us to produce build scripts that automatically generate code and integrate code fragments into a load image.

This has been demonstrated with the MetaH toolset by Honeywell. MetaH was the starting point for the AADL standard. MetaH was developed by Honeywell under DARPA funding and was used in a number of pilot projects. In cooperation with the US Army AMRDEC they have put together a tool chain and used it on hard real-time applications with highly unstable control characteristics [Feiler 2000, McDuffie1, McDuffie2]. In that context they used a modified version of the Beacon toolset to automatically generate initial MetaH models from control system models, augmented

those models with references to the sources (control models), defined the hardware architecture in MetaH (single and multiple processor configurations), and based on bindings of the application to the hardware generated a runtime executive for the specific system. This runtime executive is quite efficient and can have a small footprint. A later DARPA MoBIES program funded project at Carnegie Mellon called TimeWeaver/SysWeaver [Rajkumar 2004, DeNiz 2006] also demonstrated this capability, including automated code generation for small footprint processors in automotive applications and a linkage to Simulink models.

The user will also specify protected address spaces in form of AADL processes. These represent fault tolerance partitions, when enforced at runtime will limit the propagation of errors and faults. If dependability is of concern the AADL Error Model Annex [AS5506/1 2006, Feiler/Rugina 2006] can be used to annotate the AADL architecture model with fault information for the purpose of reliability, availability, integrity, and fault tree analysis. Reliability models in form of stochastic process models and fault trees to support hazard analysis and FMEA can be generated from the AADL model and error model annotations automatically, taking advantage of the connectivity and binding information in the AADL model to determine all possible error propagation paths. Such a capability was already supported in the MetaH toolset and is being included in the Open Source AADL Tool Environment (OSATE).

Simulink Model of Integrated System

People have used Simulink as an architecture modeling tool. Subsystem blocks can be used to represent general components. Variation of these components can be represented by configurable subsystem blocks.

Similarly, projects may integrate models of individual subsystems (e.g., throttle control, brake system, cruise control) into a single system model. This can be done in Simulink by combining the subsystem blocks representing the application subsystems appropriately and by doing the same for the models of the physical plant components.

In this use scenario we have Simulink blocks that may have been specialized to represent concurrent tasks and attributed with execution information.

One possible way of proceeding in such a setting is to define a mapping of the blocks defined in Simulink for runtime architecture modeling into the concepts of AADL. One can consider Simulink to be the graphical front-end for creating and maintaining runtime architecture. These models are then translated into AADL, either textual or through the AADL XML interchange format. This allows projects to take advantage of analysis tools that have been developed for and interfaced with this industry standard notation for modeling embedded systems.

An alternative approach is to migrate to AADL for the maintenance of the runtime architecture model, possibly having a translator map the AADL model into this library of Simulink blocks, such that users can view these models with a user interface they are familiar with.

The Simulated Plant

The AADL allows the user to specify the computing platform on which the embedded system software runs, i.e., the processors, memory, and bus/network. AADL also allows the user to represent the physical plant that the embedded system interfaces with. The AADL device component allows the user to specify a physical connection between a device and the computing platform components. It also allows the user to specify a logical interface to the application software through ports/connections, shared access, and subprogram service calls.

Devices can represent individual sensors and actuators, or they may represent the whole plant with ports representing sensor and actuator points. The device can have a set of properties that allow the user to associate references to different representations of the physical plant. In the context of a simulation run a device implementation may be defined that points to the simulation software of the plant and to different data sets to be used as the data stream generated by the device. In case of hardware in the loop execution of the embedded system, a different device implementation may carry properties that identify the driver software and any initialization parameters for the hardware, such that an appropriate build can be generated.

References

1. [SVM 2004] System Verification Manager, A DARPA MoBIES funded project led by Prof. B. Krogh at CMU/ECE, <http://www.ece.cmu.edu/~webk/svm/>
2. [Krogh 2003] Bruce H. Krogh, Peter H. Feiler, Shiva Sivashankar, and Bill Aldrich, "SVM: System Verification Manager for Model-Based Development of Embedded Control Systems", Proceedings of EMSOFT, July 2003.
3. [Krogh 2004] Bill Aldrich, Ansgar Fehnker, Peter H. Feiler, Zhi Han, Bruce H. Krogh, Eric Lim, and Shiva Sivashankar, "Managing Verification Activities Using SVM", Proceedings of Sixth International Conference on Formal Engineering Methods (ICFEM), Nov 2004.
4. [Emmeskay 2006] John Absmeier (Delphi Corporation), Tuhin Das, Swaminathan Gopalswamy, Ravi S. Paike (Emmeskay, Inc), "Development of an Automated Control System Verification Platform for a Solid Oxid Fuel Cell", Proceedings of the 4th International Conference on Fuel Cell Science, Engineering and Technology (Fuel Cells 2006), June 2006, Irvine, CA.
5. [Feiler 2000] Peter H. Feiler, Bruce Lewis, Steve Vestal, "Improving Predictability in Embedded Real-Time Systems", Software Engineering Institute, CMU/SEI-2000-SR-011, 2000, <http://www.sei.cmu.edu/publications/documents/00.reports/00sr011.html>.
6. [Feiler 2006] System Configuration With AADL, P. Feiler, Technical Report, draft, 2006.
7. [Rajkumar 2004] <http://www.escherinstitute.org/Plone/tools/basic/timeweaver>.
8. [DeNiz 2006] Dionisio de Niz, Gaurav Bhatia, and Raj Rajkumar, "Model-Based Development of Embedded Systems: The SysWeaver Approach", Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06).

9. [AS5506/1 2006] The SAE Architecture Analysis & Design Language (AADL) First Annex Collection, Peter H. Feiler (co-author), SAE International Document AS-5506/1, June 2006.
10. [Feiler/Rugina 2006] Error Modeling With AADL, P. Feiler, A. Rugina, Technical Report, draft 2006.
11. [McDuffie1] Using the Architecture Description Language MetaH for Designing and Prototyping an Embedded Spacecraft Attitude Control System, James H. McDuffie, Coleman Research Corporation.
12. [McDuffie2] Using the Architecture Description Language MetaH for Designing and Prototyping an Embedded Reconfigurable Sliding Mode Flight Controller, James H. McDuffie, Coleman Research Corporation.