Software Engineering Institute

Developing AADL Models for Control Systems: A Practitioner's Guide

John Hudak
Peter Feiler

July 2007

TECHNICAL REPORT
CMU/SEI-2007-TR-014
ESC-TR-2007-014

Dynamic Systems Program

Carnegie Mellon

Table of Contents

List of Figures

Abstract

This document is a guide to help practitioners using the Architecture Analysis and Design Language (AADL), an international industry standard for the model-based engineering of real-time and embedded systems. The primary goal of this document is to describe an approach for and the mechanics of constructing an architectural model that can be analyzed based on the AADL. The first section of this document presents an overview of AADL concepts and many of the keywords of the language. The second part of the document illustrates a model-building approach using the AADL. It takes the perspective of an engineer who is developing a model for the first time using the AADL. This guide leads the reader through complete AADL model development based on automotive embedded control systems (cruise control, traction control, etc.) by describing the use and syntax of the AADL and interleaving modeling abstraction tradeoffs to achieve models that are abstract but precise. Models are constructed with different analysis perspectives in mind to illustrate the semantics as well as the richness of the AADL.

1 Introduction to the SAE AADL

The Society of Automotive Engineers (SAE) Architecture Analysis and Design Language (AADL) is a textual and graphic language used to design and analyze the software and hardware architectures of embedded and real-time systems for performance-critical characteristics (e.g., end-to-end latency, schedulability, and reliability). The language is used to describe the structure of such systems as an assembly of software components that are mapped to an execution platform. The language is based on the component-connector paradigm. Component descriptions are used to describe functional interfaces of components (e.g., data inputs and outputs, event inputs and outputs) and performance-related aspects (such as latency and error handling). Connection characteristics (i.e., asynchronous or synchronous) among software components have precise operational semantics and are enforced by the AADL modeling environment. Dynamic behavior of the runtime architecture can also be described by supporting the modeling concepts of operational modes and transitions. Software components can be bound to hardware platforms. The hardware platforms can be modeled to reflect specific characteristics of a processor, bus, and memory.

The language is designed to be extensible to accommodate analysis of the runtime architectures that the core language does not completely support. Extensions can take the form of new properties and analysis-specific notation that can expose system variables needed by various analysis tools particular to the investigation at hand.

1.1 STRUCTURE OF THIS DOCUMENT

This document is organized to achieve multiple goals. The primary goal is to show how to develop and document system architectures using the AADL. To accomplish this goal, several aspects of the approach modeling with AADL are detailed in this document:

- system analysis and decomposition, analysis views, and AADL language constructs to model components
- the mechanics of assembling modeled components
- binding modeled software to modeled hardware
- the AADL structure of a complete model

The intent is for the reader to become knowledgeable enough to know what AADL constructs to use in assembling a model and to know how to assemble the model that can be successfully parsed and analyzed.

Another goal of this document is to introduce enough of the AADL semantics and syntax so the user can precisely represent the system to be analyzed. This goal is accomplished by illustrating in detail the complete modeling of a realistic system.

To meet these goals, the document focuses on understanding and modeling a problem in the automotive control domain. The second section provides an overview of the AADL and contains conceptual descriptions of the AADL. Readers may want to supplement this section by reviewing the AADL standard [SAE 04] and the technical note published by the Software Engineering Institute [Feiler 06a] for details about exact syntax and allowable keywords and their semantics.

The third section describes an example control problem in the automotive area that is concerned with the control and performance of a vehicle. Several analysis views of the system are presented by identifying a top-level abstraction and subsequently decomposing it into less abstract representations. The third section is divided into subsections that describe the decomposition:

- Traction control, stability control, antilock braking, and cruise control are identified and their functionality discussed.
- A top-level AADL context diagram is developed, and AADL models of the components are constructed. In addition, the operational, interactive, and performance issues related to the subsystem are discussed.
- The cruise control system is modeled to analyze latency requirements and actual latency (via flows) from the time the brake pedal is depressed to the time the throttle actuator is disengaged. Cruise control components are modeled in detail.
- System implementation is described with respect to binding modeled software components to a processor, memory, and bus.

In addition, there are several themes that cut across each of the subsections described above. These include

- producing models that represent different levels of abstraction
- describing the use of AADL keywords, semantics, and usage to enhance analysis
- describing signal flows at different levels in the software architecture
- understanding component declarations and implementations
- producing models that can be analyzed using the Open Source AADL Tool Environment (OSATE) developed at the Software Engineering Institute (SEI)

The use of OSATE is not described herein, but the models that are contained in this document have been successfully parsed and analyzed using OSATE V1.4.6, which is based on the AADL standard V1.0.

1.2 NOTATIONS USED IN THIS DOCUMENT

Text in `courier` font denotes code examples; **`courier bold`** font is used to denote AADL reserved words. Text in *italic* denotes concepts that have specific meaning within the AADL (e.g., *declaration*). The graphical AADL notation is introduced with an example as it is used. The complete set of graphical notation symbols is in Appendix A (beginning on page 61).

1.3 READER BACKGROUND

The intent of this document is to convey how to develop and use AADL models using real-world engineering examples that are typically encountered in embedded system design. By its nature, embedded system design requires a working knowledge in a number of different areas. It is therefore helpful for the reader to have a background in the following areas:

- system decomposition and modeling—use of models to represent physical systems at the right level of granularity to analyze a problem domain
- control engineering—basic notion of feedback control and realizations in the continuous and discrete domains
- real-time operating systems—understanding of process, thread, and timing concepts as well as data and event communication concepts

In addition, an exposure to the SAE AADL is helpful. It is expected that the reader has an understanding of the concepts and constructs of the AADL (from introductory papers or, preferably, from reading through the standard) [SAE 04]. This document presents a high-level overview of the language in Section 2 but not the exact syntax. The examples developed in Section 3 show the use of the language in constructing models, but familiarity with the language would be very helpful. One major goal of this document is to help the reader define and assemble the pieces of an AADL model, so having a copy of the standard would be beneficial.

1.4 BACKGROUND OF THE AADL

The AADL is based on experiences in the use of Defense Advanced Research Projects Agency (DARPA)-funded Architecture Description Language (ADL) projects, MetaH in particular. MetaH was developed by Honeywell [Binns 96] and has been used by a number of organizations in prototypical system developments, including Boeing, the U.S. Army, and the SEI. The case study of a pilot application of the MetaH technology by the U.S. Army's Armament Munitions and Chemical Command Software Engineering Division (AMCOM SED) laboratory to missile-guidance systems produced some insight into the potential cost savings of an architecture-driven modeling approach. An existing missile-guidance system, implemented in Jovial, was reengineered to run on a new hardware platform and to fit into generic missile reference architecture [McConnell 96]. As part of the reengineering effort, the system was modularized and translated into Ada95. The task architecture consisting of 12 to 16 concurrent tasks was represented as a MetaH model and the Ada95 coded application components. The cost savings ranged from 50% for reengineering to a different language and platform to 90% for porting to another platform.

MetaH demonstrated the practicality of using an ADL with precise real-time semantics and syntax as a core modeling notation for providing analysis capabilities of several performance-critical quality attribute dimensions, such as schedulability, dependability, and safety-critical concerns. The MetaH tool set demonstrated the capability of not only supporting system analysis but also automatic generation of glue code in the form of a system executive that performs all task binding, dispatching, and inter-task communication with application components as "plug-ins" into this infrastructure. This separation of concerns allows application developers to focus on domain functionality, while a software system architect can focus on achieving system-level performance-critical quality attributes. MetaH has demonstrated the feasibility of model-based analysis and generation of embedded systems (i.e., embedded and system-of-systems engineering driven by an architecture that is reflected in the models and maintained throughout the system life cycle [Feiler 00]) and has thus served as a proof of concept for the AADL.

2 An Overview of the AADL

The AADL was designed to be a basis for model-based analysis and generation of embedded systems. The AADL notation was designed as an extensible core language with well-defined semantics and both a graphical and textual presentation. The core language supports modeling in several architectural views [Clements 02] and addresses timing and performance analyses through explicit modeling of the application system and binding to execution platform components. Timing and performance analyses are modeled through precisely defined concurrency and interaction semantics and timing/performance properties that are explicitly associated with the components.

This section introduces the concepts of the core language. A major subset of the AADL keywords (in bold) is identified and their meaning presented. The reader is referred to the AADL specification for a more detailed description of the language [SAE 04]. Section 3 of this report builds on the keywords by first showing generic usage syntax and semantics and then by developing an example problem throughout the section. Several analysis views are also presented to illustrate the use of specific AADL concepts (e.g., data communication and signal flow).

The focus of the AADL is to model the software system architecture in terms of an application system bound to an execution platform. Most of the modeling effort is directed to the application-specific software, incorporating just enough operating system specifics to ensure that the model is precise enough to support the investigation adequately. Many operating system characteristics are already embedded in the AADL semantics to support architectural modeling (communication mechanisms, thread behavior, etc.).

The AADL language syntax is composed of keywords that represent basic components, connections, and behavior. The keyword often contains additional keywords that describe features, properties, refinements, or extensions to the basic component. The software architecture is modeled in terms of components and interactions. A component represents some hardware or software entity that is part of a system being modeled in AADL. The standard defines the following categories of components: **`data`**, **`subprogram`**, **`thread`**, **`thread group`**, **`process`**, **`memory`**, **`bus`**, **`processor`**, **`device`**, and **`system`**. These are explained later in this section.

A *component type* specifies a functional interface of the component. It can be viewed as a "black box" in which only the external interfaces are described. It represents a specification of the component against which other components can operate. To satisfy the requirements for an AADL specification, one must specify *component implementations*. A component implementation specifies an internal structure (e.g., the operational details or "white box"). The AADL allows multiple implementations to be declared with the same functional interface. Interactions of components are specified through named connections and what is being communicated through the connections—either state or event information.

Components are named and have a component type that represents the component's externally visible interface and other characteristics (i.e., represents a component specification). A component type declaration may contain subclauses that represent **`features`**, property associations (**`properties`**), and **`flows`**. A feature of a component models a component characteristic that is visible to other components. Features are externally visible, named parts of the component type and are used to exchange control and data via connections with other components. Properties are expressions that represent attributes (and possibly associated values) and behaviors of a component (i.e., **`modes`**). Modes are described later in Section 2.4.

Component type can be declared in terms of other component types (i.e., a component type can extend another component type, inheriting its features, flows, and property association). If a component type **`extends`** another component type, then features, flows, and property associations can be added to those already inherited. A component type extending another component type can also refine (**`refines`**) the inherited features and flow declaration by more completely specifying partially declared component classifiers and by associating new values with properties. For example, an **`in data port`** feature of a component type may be refined to denote a specific data type (e.g., bool). Section 2.2 describes port types.

A component implementation in the AADL specifies an internal structure for a component as an assembly of subcomponents. Component implementations can be decomposed into an interconnected set of subcomponents that refer to instances of other component types and implementations. Specialization of components is supported in that component types may be extended by using **`extends`** and component implementations may be further clarified by using **`refines`**. Components must be declared before they can be instantiated or referenced.

The AADL supports modeling of the system at various levels of abstraction, allowing the modeler to quantify certain aspects of system behavior by focusing on various performance characteristics (e.g., end-to-end latency, modal operation, and reliability). The general modeling approach is to define system components that will eventually map into application code via semantic properties. This approach implies using model variables that have the same name within the application code. Matching of variable names throughout the development process can be accomplished manually or as a result of using automatic code generators that obtain the variable names from the AADL models. Ensuring correct variable mapping from model to actual code is necessary to ensure consistency.

2.1 APPLICATION SYSTEM COMPONENTS

Application system modeling is supported through a number of groups of component categories. The first group focuses on the runtime behavior of a system: a **`thread`** is a basic unit of concurrent execution; a **`process`** is a unit of protected address space. Threads are contained in processes and have a dispatch protocol property value. A process must contain at least one thread. Predefined dispatch protocol values include periodic, aperiodic, sporadic, and background. Additional protocol names can be declared using the property extension mechanism. Threads are characterized by entry points. They have separate execution entry points into their associated source text for initialization, nominal execution, and recovery. Nominal execution server threads may have separate entry points for each server subprogram. The process load, thread dispatch, and scheduling semantics are defined using a hybrid automaton notation that is specified as part of the AADL.

Threads contained in a process may be logically organized into a hierarchy using thread groups. A **`thread group`** *type* declares the features and required subcomponent access through which threads contained in a thread group can interact with components declared outside the thread group. Features and subcomponent accesses defined in the thread group declaration are inherited by the **`thread group`** *implementation*. The thread group implementation contains threads, data components accessed by the thread, and thread groups.

Software systems to be modeled can be organized into a hierarchy of AADL components to reflect the logical flow and physical bindings of the application. The system construct is a generic component hierarchy that is modeled by **`system`** *declarations* to represent a composition of components into composite components (e.g., a device communicating data to another component). A system instance models an application and its binding to a system that contains execution platform components (e.g., a thread bound to memory connected to a bus to a central processing unit [CPU]).

A **`system`** can be used to organize processes, execution platform components, or the combination of both. A **`system`** can also contain lower level system instances. It is generally used early in the modeling process as a generic modeling component and can be made more specific when decisions are made about

- which components will be included within the system
- how they will compose the workload via units of concurrent execution
- how they will employ address-space protection.

Similarly, a process is composed of threads and thread groups.

The **`data`** component category supports representing data types and data abstractions in the source text at the appropriate level of abstraction for the modeling effort. The **`data`** *type* is used to type ports (ports are the interface points to the outside, as described in Section 2.2) to specify **`subprogram`** parameter types. Data type inheritance can be modeled using the AADL component type extension (e.g., **`extends`**) mechanism. Class abstractions can be represented by using **`subprogram`** as a feature of a data type. Provided data features of a data component are sharable using a specified concurrency control property. A sharable data component instance is specified in a **`requires`** subclause of the component type.

2.2 CONNECTIONS AND PORTS

Connections among components are supported in three ways:

1. directional flow of data and control through **`data`**, **`event`**, and **`event data port`** connections
2. call-return interaction on **`subprogram`** entry points
3. access to shared **`data`** components

Threads, processors, devices, and their enclosing components (processes, systems, and thread groups) can have ports (**`in`**, **`out`**, **`in out`**) declared. A **`data port`** communicates unqueued state data; an **`event port`** communicates events that are raised in a component's implementation, associated source text, or actual hardware; and an **`event data port`** represents queued data whose arrival can have event semantics. Arrival of an event at a thread results in the dispatch of that thread using semantics defined via thread property values and the AADL's hybrid automata for event arrival while the thread is active. For **`data port`** connections, data is communicated upon execution completion or upon thread deadline. For periodic threads, data communication upon completion of the thread execution is an immediate connection with the effect of midframe communication threads. The case where data is communicated upon a thread deadline represents a delayed connection with the effect of phase delay. Threads are discussed in Section 2.7.

2.3 MAPPINGS TO SOURCE TEXT

Software components model source text where source text is the target programming language. The target programming language can be general languages such as C and Ada95 or domain-specific languages such as Simulink. Rules governing the mapping between the AADL model and source text depend on the applicable programming or modeling language standard. Predeclared AADL component properties identify the source text container and the mapping of AADL concepts to source text declarations and statements. AADL port variables, data components, subprogram, and entry points allow mapping of AADL names to names in source text.

2.4 MODES

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

2.5 FAULT/EXCEPTION HANDLING

The AADL has a basic fault-handling model. Runtime faults may be handled within source text components through mechanisms that are part of the source language runtime environment (i.e., an application fault handler, fault detection, and handling of the application language). For faults not handled at that level or propagated by the source text, a **`thread`** is given an opportunity to recover and continue with the next dispatch through a recovery entry point. An unrecoverable thread error is propagated through a predeclared **`event data port`** called `error`. The modeler of a particular application system indicates through an event connection specified in the AADL model where the error event is propagated to, and mode change behavior descriptions indicate the actions to be taken in response to error events [Feiler 07].

2.6 FLOW SPECIFICATIONS

The AADL supports the concept of specifying end-to-end flows to support various forms of end-to-end analysis throughout a model such as end-to-end timing and latency, reliability, numerical error propagation, and processing sequences of domain objects. System, process, and thread components can have flows. A flow specification declaration indicates that information logically flows from one of the incoming ports, parameters, or port groups of a component to one of its outgoing ports, parameters, or port groups. The ports can be event, event data, or data ports. A flow may start within a component, designated as a **`flow source`**, and end within a component, called a **`flow sink`**. A **`flow path`** through a component can be from one of its **`in`** or **`in out`** ports or parameters to one of its **`out`** or **`in out`** ports or parameters or, in the case of port groups, from a **`port group`** to its **`inverse`**.

Component implementations must provide a flow implementation for each flow specification. A flow implementation declaration identifies the flow through its subcomponents. In the case of a flow-source specification, it begins from a **`flow source`** of a subcomponent or from the component implementation itself and ends with the port name in the **`flow source`** specification. In the case of a flow-sink specification, the flow implementation starts with a port name in the flow-sink specification declaration and ends within the component implementation itself or within a **`flow sink`** of a subcomponent. Flow characteristics modeled by properties on the flow implementation are constrained by the property values in the flow specification. Flow implementations can be declared to be mode specific.

2.7 EXECUTION PLATFORM COMPONENTS

The AADL supports four categories of execution platform components: **`processor`**, **`memory`**, **`bus`**, and **`device`**. A **`processor`** is an abstraction of the hardware and software that is responsible for scheduling and executing threads. For their execution, threads will be bound to a processor that supports the dispatch protocol required by the thread. The `Allowed_Dispatch_Protocol` property specifies the dispatch protocol that a processor supplies. Processors may also execute device driver software associated with a device, but they must have access to the corresponding device component. Processors may contain **memory** and may access memories and devices via buses. A **processor** component must contain at least one **memory** component (the application software executes in the attached memory) or provide access to memories via a **bus**.

A **memory** declaration represents an execution platform component that stores binary images. Memory can represent any randomly accessible storage (e.g., random access memory [RAM], read-only memory [ROM]) or more complex (random or sequential) storage devices (e.g., magnetic disk, optical disk, and tape). Memories have **properties** such as size (e.g., 8 bits), protocol (e.g., `read_only`, `write_only`, `read_write`), and amount (e.g., `max_word_count`). Subprograms, data, and processes are bound to **memory** components for access by processors when executing threads. A **memory** component may be contained within a **processor** declaration or may be accessible from a **processor** via a **bus**. This accessibility is realized by the **access** keyword in the **processor** declaration.

A **bus** provides access between processors, memories, and devices. The **bus** component represents a communication channel, typically hardware coupled with a communication protocol. Typical examples of a bus are Peripheral Component Interconnect (PCI) and Ethernet (category 5 cable), over which runs Transmission Control Protocol/Internet Protocol (TCP/IP). Processors, memory, and devices can communicate over a shared **bus**. The shared **bus** can be located in the same **system** implementation as the execution platform components sharing it or higher in the system hierarchy via **access features** in the system types.

Memory, **process**, and **device** types can declare a need for access to a **bus** through a **requires bus access** reference. The `allowed_connection_protocol` property indicates which forms of access a particular **bus** supports. A **device** is accessible from a **processor** if the two share a **bus** component and the `allowed_connection_protocol` property value for that **bus** includes the enumeration literal `device_access`. A **memory** is accessible from a **processor** if the two share a **bus** component and the `allowed_connection_protocol` property value for that **bus** includes the enumeration literal `memory_access`.

A **device** represents a component that interfaces with the external environment. A **device** can represent single-function components (e.g., sensors) as well as more complicated components (e.g., global positioning system [GPS] units, camcorders). In reality, the more complicated devices would have internal processors, memory, and so forth that are not explicitly modeled. For example, the modeling of a camcorder might consist of only a connection to a firewire bus over which video data is communicated. Data is communicated out of (or into) the device by configuring it with **out data port** or **in data port**. If the device has associated software (such as a device driver that must reside in a memory and execute on a processor external to the device), it can be specified through appropriate property values for the device (e.g., `source_code_size`, `source_code_data`).

A **device** interacts with both AADL execution platform components and application software components. A **device** has physical connections to processors via a bus. The **device** modeling element represents software executing on a processor accessing the physical device. A **`device`** also has logical connections to application software components. These logical connections are represented by connection declarations between device ports and application software component ports. For each logical connection between a device and a thread executing application source text, there must be a physical connection in the execution platform.

2.8 MODEL ORGANIZATION

The AADL specification is a set of declarations: component classifiers, port group classifiers, annex libraries, packages, and property sets. A **`package`** provides a way to organize component types, component implementations, port group types, and annex libraries into related sets of declarations by introducing separate name spaces. The name space is divided into **`public`** and **`private`** parts. Items declared in the **`public`** part are visible and can be referenced from outside or inside the **`package`**. Items declared in the **`private`** part can be referenced only within the **`package`**.

A **`property`** provides information about component types, component implementations, subcomponents, features, connections, flows, modes, and subprogram calls. A **`property`** has a name, type, and value. A **`property set`** contains declarations of property types and property names that may appear in an AADL specification. The AADL contains two predeclared property sets in the standard (`AADL_Properties` and `AADL_Project)` that define properties and property types that are applicable to all AADL specifications. Users may define property sets that are unique to their model or project.

2.9 SUMMARY

The AADL standard supports modeling of application systems and execution platforms as interacting components with specific semantics and bindings. Such systems are configurable in that components can have multiple implementations. Semantics defined as part of the component categories and their predefined properties address timing and resource consumption as well as interaction consistency in terms of matching port types and data communicated through the ports. Behavioral descriptions allow for model checking of behaviors as well as mode (state)-specific analyses with less conservative results.

The core language does not provide properties and semantics for all possible architecture analyses. Instead the AADL has been made extensible both in terms of language notation and standard annexes to accommodate further analyses. Annexes that are part of the initial core standard include language extensions such as flow specification, the ability to provide an error model for reliability analysis, and the ability to support end-to-end flow analysis. Other annexes include a Unified Modeling Language (UML) profile and an Extensible Markup Language (XML) interchange format.[1] [jmorley1]An open source development environment (OSATE) based on Eclipse

[1] Annexes for error modeling, programming language compliance and application program interface (API), and XML/XMI interchange format have been published as extensions to the standard language [SAE 06].

that contains a parser and some analysis plug-ins is available for download at http://www.aadl.info/.

3 Developing Models Using the AADL

First-time developers of AADL models typically have many questions regarding model organization, composition, and the mechanics of assembling a model. There are also issues related to determination of the analysis perspective and problem details that can sometimes result in a struggle when building a model. This section addresses those issues by describing a model-building process that focuses primarily on expressing system architectures in terms of the AADL constructs; it also shows how certain architectural issues can be modeled at the proper level of detail. (See *Improving Predictability in Embedded Real-Time Systems* [Feiler 04] for a discussion related to this topic in modeling an existing architecture.) The process also shows that certain design decisions, such as a task-to-thread binding or a thread-to-processor binding, can be delayed by being pushed down into the lower levels of the model hierarchy without adversely affecting the fidelity of the higher level model. Through the cruise control example, details relating to information flow and end-to-end latency are discussed and modeled (see Section 3.2).

A general outline for problem analysis and model development involves the following steps:

1. Determine the scope of the area to investigate (Section 3.1).
2. Determine the perspective (e.g., the analysis view) (Section 3.2).
3. Understand the system components and their functionality (Section 3.2).
4. Map the functional components and the data and event flows to AADL components and connections (Sections 3.3−3.4).
5. Direct the output of the AADL parser to the appropriate analysis tool and evaluate the results (Section 3.5).

These steps are followed as we develop the example model in this section. We must first establish the example problem.

3.1 AN AUTOMOTIVE EXAMPLE PROBLEM

To help in the understanding and application of the AADL, an example problem based on automotive control systems is analyzed. The first step in problem analysis is to determine the scope. Consider the following automotive systems:

- traction control system (TCS)
- antilock braking system (ABS)
- stability control system (SCS)
- cruise control system (CCS)

These systems represent a small number of embedded systems contained in modern automobiles. They are an interesting subset of systems to consider mainly because the hardware and software architecture of a solution can have many approaches, ranging from a single CPU executing the application code for all of the systems, to a separate CPU for each system. The architecture may take into account a partitioned system and various fault-tolerant schemes. We initially focus on identifying the components of each system and then choose a specific subsystem for more detailed modeling and analysis.

In general, each of these systems relies on input from sensors, makes computations based on its specific control laws, and outputs a control value that affects the state of the engine, throttle position, or the brakes.

We will eventually choose one of these subsystems to focus on, but initially, their collective operation must be understood. The following paragraphs present functional descriptions from which system architects could begin modeling. The descriptions could be viewed as part of the system's requirements. As part of this example, we show how to go from descriptions of systems with a varying amount of detail to architecture models represented in the AADL that allow modeling at many levels within a system hierarchy. We also suggest the types of analysis and modeling tradeoffs that can be made at each level of detail. At this point in the example, we are not making any assumptions regarding the number of processors required or the mapping of the control systems to processors. Later in the development of the example, we will show the mapping to a single processor.

The traction control system deals specifically with lateral (front-to-back) loss of tire to road friction during acceleration. In its most basic version, the traction control system uses the data from the rotation sensor of each wheel of the vehicle, compares the rotation data to detect the slipping wheel(s), and compensates for the spinning wheels by applying the brake to the spinning wheels to ensure maximum contact between the road surface and the tires, even under less-than-ideal road conditions, such as ice or snow. In some cases, the traction control system may even reduce the power to the engine via the throttle control.

The goal of the antilock braking system is to ensure that maximum braking is accomplished at all four wheels of the vehicle, even under adverse conditions such as skidding on rain, snow, or ice. Antilock brakes work by sensing slippage at the wheels during braking and continually adjusting braking pressure to ensure maximum contact between the tires and the road. In the most basic version, wheel rotation sensors from all four wheels are used as input and the output is the brake valve on each of the four brake lines.

The intent of stability control is to keep the vehicle going in the direction in which the driver is steering the car. To do this, the stability control system applies the brakes to one wheel to help steer the car in the correct direction. For example, if poor traction causes the front end of the car to slip sideways when you are going around a corner (understeering or plowing), the control laws will cause the brakes to be applied on the inside wheels of the corner, causing the car to turn and slow down. If the back end of the car slips sideways (oversteering or fishtailing), the brake on the wheel that is outside of the corner is applied to bring the car back into line. The system works when the car starts to slide on a straight road and when turning corners.

The stability control system controller takes information from a number of sensors and then determines whether the car is in a stable or unstable state. By combining the data from wheel rotation sensors, steering angle sensors, yaw sensors that measure the amount a car rotates around its vertical center axis (fishtails), and lateral force sensors that measure the amount of sideways g-force generated by the car, the central processing unit can actually detect when a vehicle is behaving in a way contrary to how the driver intends. The control system then applies the brakes to the wheel(s) to counteract the destabilizing force. In some cases, the engine speed may also be adjusted.

How does a stability control system differ from a traction control system? Traction control acts on a vehicle's drive wheels to prevent unwanted wheel spin under acceleration. While this helps in low-traction situations such as snow or rain, traction control's ability to assist in more extreme emergency situations is limited. A stability control system, on the other hand, goes one step further by actually detecting when a driver has lost some control. It then automatically stabilizes the vehicle to help the driver regain control of the vehicle.

The goal of the cruise control system is to maintain a constant vehicle velocity as determined by a driver-dictated setpoint. The system is in effect between some minimum and maximum speeds (e.g., 35 MPH to 100 MPH). The cruise control system maintains the vehicle speed at the predetermined setpoint by noting the speed of the wheel rotation when the setpoint is set and attempts to keep the throttle actuator at a position to maintain the vehicle speed at the speed setpoint. As the road inclination changes, the vehicle speed is affected, and the throttle position changes to maintain the vehicle speed in accordance with the speed setpoint. The control system observes the speed difference between the current speed and speed setpoint and either decreases or increases the throttle actuator position to counteract the speed differential. The algorithm to accomplish this is called the control law.

In all of the above systems, signals regarding overall engine and vehicle state (e.g., engine on/off, brake pedal on/off) are also considered in each control subsystem. These signals are used to ensure proper operation of each subsystem. For example, if the brake pedal is depressed, the cruise control system should disengage (or not become active) and the traction control system should not become active. There are also outputs from each system used as feedback to the driver that each system is on. These outputs can be indicating lights or some form of intelligent operator display (e.g., LCD panel).

The systems described above are illustrated in Figure 1.

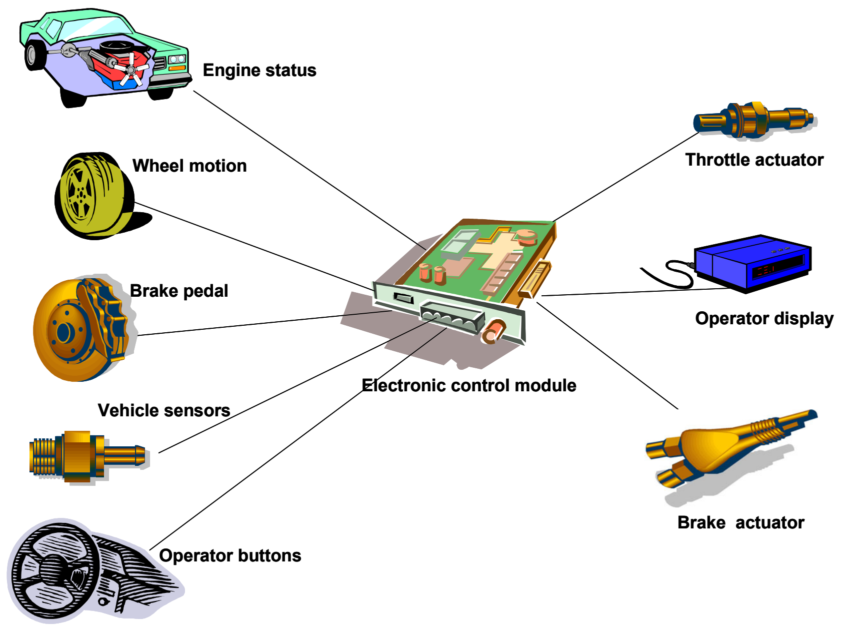


Figure 1: Components of a Number of Vehicle-Control Systems

On the left side of Figure 1 are representations of the different inputs that feed a controller (or controllers). The right side shows the output devices that receive control information or status information for display purposes. For modeling, a context diagram is needed to show the system components and their associated interactions (i.e., the structure of the system). In this particular case, we are interested in the signal processing performed by the system and the flow of the control and status information. Figure 1 can be transformed using the previous descriptions and represented as a context diagram using the AADL graphical notation. The AADL graphical notation is proposed for this purpose because it can adequately represent the systems and components at a high level and convey some information about component behavior, variable definition, and flow for purposes of analysis. The intent of this diagram is to show the model components and external interfaces that map directly to component declarations within the AADL textual model. The AADL has the capability to model systems and components of a hierarchical structure to whatever level of detail is necessary to evaluate various aspects of system performance, such as end-to-end latency and redundancy. The AADL "context" diagram is shown in Figure 2.

This top-level model makes use of the AADL system (e.g., traction control system), device (e.g., engine), and port group constructs. At this level of abstraction, we know that data provided by the physical components will be going to each of the various application components. Two modeling decisions have been made at this point:

1. show the physical components as devices
2. notationally bundle the data provided by each physical component as a port group (the specific data of interest can be separated out later)

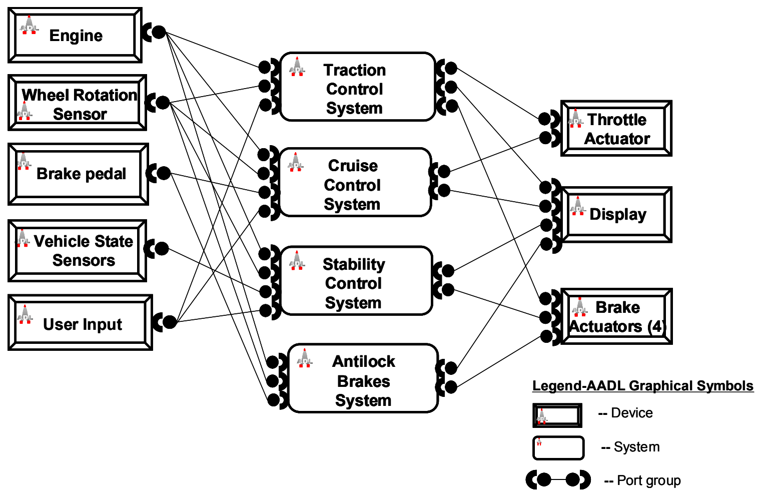


Figure 2: AADL Context Diagram for a Set of Vehicle Control Systems

The physical components are represented as AADL devices. The **`device`** construct is generally used to model sensors. Sensors can be simple or complex. The device construct allows the inner workings of the components to be hidden and only exposes the data that is to be communicated. For example, the wheel information that is required by the various applications is revolutions per minute (RPM). The wheel-rotation sensor device produces pulses, which over time can be integrated to indicate speed. The wheel-rotation sensor device is described by the following declaration:

```
device wheel_rotation_sensor
   features
      wheel_signal:  port group wheel_sensors_socket_1;
end wheel_rotation_sensor;
```

For the **`device`** `wheel_rotation_sensor`, we must specify the data that it presents to the outside world within the **`features`** declaration. In this example, the **`port group`** concept was chosen to bundle the variables. Section 3.3.2 shows an alternative modeling approach that declares each variable within the **`device`** declaration. The graphical icon for the **`port group`** connector on `wheel_rotation_sensor` is a semicircle with a ball in it. There is a matching symbol at the other end of the connecting line. One can think of the semicircle as being the collecting point of all the variables within the device and emanating out of the device as a group. The variable updates are independent of one another and are not determined until **`thread`** assignments are made. Timing relationships are discussed in Section 3.4. The interface outside of the wheel rotation sensor is declared in the variable list of the `wheel_sensors_socket_1` **`port group`**:

```
port group   wheel_sensors_socket_1
   features
         wheel_pulse: in data port bool_type;
end wheel_sensors_socket_1;
```

Each variable is declared (e.g., `wheel_pulse`) as well as the direction and the type (**`data port`** or **`event data port`**). The variable `wheel_pulse` could have been represented as an event data port. The type depends on whether the signal is to be sampled or event driven. We are assuming a sampled system.

We also made use of variable typing that is part of the AADL. Variable typing is used to enforce type consistency between components. For this example, it is added to the end of the variable port definition and is of type Boolean. Type checking along connections is performed by the AADL parser. Type checking is performed on the model where both ends of a connection declaration must have matching types. When the model is instantiated (discussed in Section 3.5.2), the type consistency also applies to semantic connections. This checking is satisfied if the declarative model is consistent.

Additional information relevant to analysis may be added to the wheel model as required. In the discussion of the traction control system, it is stated that the application would detect wheel slippage to better control the vehicle. The wheel device makes the measurements and computations to provide an indication of slippage. Those details of the computation are not important at this level of modeling; just the indication of slippage is necessary. Presumably, the wheel device contains an embedded CPU that performs the computations necessary to produce a wheel slippage value or event. The wheel device declaration above is modified to reflect the inclusion of slippage information.

```
port group   wheel_sensors_socket_1
   features
         wheel_pulse: in data port bool_type;
         wheel_slippage: in data port real_type;
end wheel_sensors_socket_1;
```

Slippage can be computed as an angular acceleration. At a constant rotational velocity, the angular acceleration would be zero. When the wheel slips, the angular acceleration would be greater than zero, indicating some relative degree of slippage, hence the typing of the `wheel_slippage` variable as real. For a wheel to be slipping, the angular acceleration would have to be greater than some predetermined value.

The port group concept is based on the notion that variables are gathered at the external interface of a component and then split apart at the interface of another component, where they are acted on internal to that component. The `wheel_sensor_socket_1` declaration above illustrates the gathering. The splitting apart notion is done by using the AADL **`inverse of`** construct:

```
port group   wheel_sensors_plug_1
   inverse of wheel_sensors_socket_1
end wheel_sensors_plug_1;
```

The semantics of **`inverse`** mean that the mating connector is declared, the variable names and associated typing remain the same, and the direction of the data is the inverse of the mating port group. The direction change is made clear in the next few paragraphs.

The remaining physical components of the automobile-control systems are modeled in the same way using the **`device`** construct. The complete set of device declarations is in the model shown in Appendix B (beginning on page 63).

The traction control, stability control, cruise control, and antilock brake applications are modeled using the generic construct **`system`**. At this level of detail, the **`system`** declaration represents the application software components and associated connections. The semantics of the system construct state that the components must be bound to an execution platform. At this point in the development of the model, decisions have not been made with respect to binding the application to an execution platform. Each system may be bound to a separate processor, bound to the same processor, or some combination. Binding decisions can be done at a later time in the modeling process. Section 3.5.1 discusses model implementation and binding for this example.

```
As an example, the system declaration of the traction control
application would look like this:

system traction_control_system
   features
      tcs_wheel_input: port group wheel_sensors_plug_1;
      tcs_engine_input: port group engine_plug_1;
      tcs_user_input: port group user_console_plug_1;

      tcs_throttle_output: port group throttle_actuator_plug_1;
      tcs_display: port group user_display_plug_1;
      tcs_brake_output: port group brake_actuator_plug_1;
end traction_control_system;
```

Note that the input variables used in the traction control system come from different devices, and, similarly, the outputs go to different devices (Figure 2). As an aid to understanding the connectivity among model components, directional hints are used in the naming of the port group variables. In this particular case, the name (e.g., tcs_wheel_input) contains the direction (input) that can be associated with the component being referenced. Declarations for the cruise control, stability control, and antilock brake systems are composed in a similar fashion and are shown in Appendix B.

Building on the above discussion regarding port groups, the variables in the engine device are used in the traction control system and bundling of the variables is accomplished using the port group:

```
port group   engine_socket_1
   features
         engine_state: out data port;-- engine is off (0) or on (1)
         engine_temp_1:  out data port;
         throttle_position: out data port;
end engine_socket_1;
```

The `engine_socket_1` **`port group`** declares three variables: (1) `engine_state`, (2) `engine_temp_1`, and (3) `throttle_position`. This grouping of ports allows the number of connection declarations to be reduced, especially at higher levels of a system when a number of ports from one subcomponent and its contained subcomponents must be connected to ports in another subcomponent and its contained subcomponents. The semantics of the **`port group`** construct state that the variables are available to the component but do not mean that they are used by the component. Components may or may not connect to the variable within a port group. The **`port group`** construct does not impose any timing-related information with respect to each variable within the **`port group`**.

Timing semantics for data communication at each port are determined by the corresponding **thread** model and **processor** binding, and **data** is valid when the **thread** execution is complete. There are two communication mechanisms that can be used: immediate and delayed. Immediate communication occurs among threads that execute within the same partition. The **data** is communicated at the end of **thread** execution and is immediately available to be used by the next **thread**, as indicated in the specified connection.

Delayed communication occurs across partitions. Data in the source partition is valid only at the end of the execution of the source partition. When the next (destination) partition begins execution, any data that is shown to be connected to the destination partition will be valid at the time of execution.

The engine socket port group consists of three outputs (`engine_state`, `engine_temp`, and `throttle_position`) that will be connected to some other component. The component to which the **port group** is being connected will have, as part of its interface, inputs that correspond to the matching outputs from the engine. To ensure this matching of the number and names of signals, the **inverse of** reserved word would be used to represent the complement to the referenced **port group** type. An example of the use of **port group inverse** declaration is shown below:

```
port group   engine_plug_1
inverse of engine_socket_1
end engine_plug_1;
```

A port group declaration named `engine_plug_1` is made to indicate that it contains a complement of the input and output ports associated with `engine_socket_1`. It should be noted that port groups support signal "fan-out" where data from an **out** data (or **event**) port can feed an infinite number of corresponding **in** data ports. The notion of "fan-in" does not make sense for data but does work for event data, because event data is queued.

Alternatively, we could have represented each data port declaration individually within the features section of each system. Graphically, each data port would have a connection from its output to each of its corresponding inputs. For example, the declaration of the traction control system would be shown as follows:

```
system traction_control_system
   features
      tcs_wheel_pulse: in data port;
      tcs_engine_on: in data port;
      tcs_on_off: in data port;

      tcs_throttle_output: out data port;
      tcs_on_off_indicator: out data port;
      tcs_brake_output: out data port;
end traction_control_system;
```

The actual connections are made when the model is instantiated. This model-building approach is discussed in Section 3.5.

Output device declarations must be made to complete the example. The device construct is used to model the displays and actuators because of the potential complexity of the components. For example, the operator display would most likely be an embedded controller, but for modeling purposes, we are interested only in what data it receives (and eventually displays), not the internal details of how it actually accomplishes the operation. The device declaration for the operator display is shown below:

```
device display
   features
        tc_display_input_signals:  port group
tc_user_display_socket_1;
        cc_display_input_signals:  port group
cc_user_display_socket_1;
        sc_display_input_signals:  port group
sc_user_display_socket_1;
        abs_display_input_signals:  port group
abs_user_display_socket_1;
end display;
```

The variables that the display actually receives are contained in the **`port group`** declaration that the display declaration refers to, specifically port groups `tc`, `cc`, `sc`, and `abs` user display sockets. The port group `tc_user_display_socket_1` is shown below as an example.

```
port group   tc_user_display_socket_1
   features
          tc_state: out data port;
end tc_user_display_socket_1;
```

The variable `tc_state` contains status information of the traction control system, namely that it is either on or off. For the operator display, the first three data ports represent the state of their respective system being on or off. For the antilock brake system, fault information is displayed. All four inputs are Boolean signals. The remaining devices are similar in their declarations and associated port group, and they are shown in Appendix B (beginning on page 63).

This example shows the use of the **`device`**, **`system`**, and **`port group`** constructs of the AADL. As shown in Figure 2 on page 17, it also illustrates the use of the graphical AADL notation to construct a context diagram from which the component declarations were derived. The context diagram shows the logical connections of the data and the data flow from the device through the application to the actuator (an analysis theme that is elaborated on later in Section 3.4). AADL diagrams at this level serve to document devices, application software, and the structural data relations between the devices and application software. Analyses that could ensue from this level of detail include correct variable mapping, specification-based computational latency of signal flow, system interdependency, and possible variable access ordering (i.e., two or more applications writing to the same device). Some of these analyses are described in detail in the following sections.

At this point in the development of the top-level model, all of the component types and their data have been defined. Figure 2 (page 17) shows the data flow from left to right. This figure shows several behavioral facts about the system as modeled in the AADL. For example, the data flows from the devices on the left, through some application software, to devices on the right. Data ports are named with variable names of interest that can eventually be mapped to the source text. Data ports represent external data interfaces of the component. In the declarative model, the data is typed. No decisions have been made about how the data will be realized (e.g., shared variable, register, local) or about the access methods (e.g., open, mutex, asynchronous, synchronous). These details will be discussed when an actual implementation model is developed and component connection semantics are considered. To represent this detail, a model of the connection and associated threads is developed. The detailed implementation model is discussed in Section 3.5. When the software component (i.e., device or system) is eventually mapped onto an execution platform, the values associated with the output ports will be available at the end of the execution time slot of the thread. In actuality, the task may complete execution before the deadline, but the data will not be presented until the deadline, assuming no preemption.

The four application software systems process data independently of one another. This independence suggests some degree of concurrency among all of the applications, which has ramifications with respect to threading, scheduling, and binding to an execution platform. Decisions concerning these implementation aspects can be delayed until late in the modeling and system development process. These decisions are discussed later in this example to show how the base model can be expanded to include the precise detail needed for system investigation.

Another set of observations can be made with respect to the output devices. The same control variable is acted on by separate applications. The brake actuators receive brake signals from the traction control system, the stability control system, and the antilock brake system. The throttle actuator receives throttle position data from the cruise control system and from the stability control system, which raises questions about the effects of possible conflicting data to the device. For example, the cruise control may be producing throttle positions corresponding to an increase of speed, while the stability control system may be calling for a decrease in speed. From a control engineer's perspective, each individual control loop would be designed, developed, and implemented as if it were the only application communicating with the devices. This assumption is reasonable when developing control systems. Furthermore, the control engineer may assume that the control system will be running on a dedicated processor and that this application will have total control and access to all resources. Although this assumption is commonly made, it may not reflect reality. From a cost perspective, it may be necessary to locate all the applications on a single processor or split the applications between two processors. While this decision does not affect the design of the control laws (i.e., the sampling frequency and stability should remain the same), it may cause errors in the operation of the integrated system due to the effects of concurrency and scheduling decisions.

Investigating the possible architectures that arise when application software is integrated onto a single processor or spread across multiple processors is outside the scope of this report. It is a scenario that is well suited for modeling and analyzing designs with the AADL. This example underscores the ways in which analysis using the AADL can reveal problems of this type that manifest themselves during integration.

3.1.1 Perspectives on Data Flow

The second step of problem analysis is to determine the perspective for analysis.

Embedded real-time systems that contain software controllers have data flow and timing issues that are viewed in one way by the control engineer (e.g., consistent sampling rate, minimal/no resource contention) and in a slightly different way by the software system engineer (e.g., multiple tasks and scheduling issues, resource utilization).

Figure 2 on page 17 showed an abstraction of the vehicle-control system from a systems engineer's viewpoint and captured the essence of the solution. Figure 3 shows the control system from a control engineer's viewpoint.

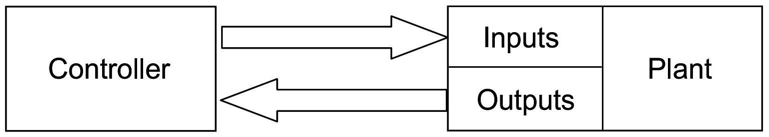


Figure 3: A Control Engineer's View of a Control System

The actual physical plant is shown on the right side of Figure 3. In our example, the physical plant is the automobile (e.g., engine or steering) and the control variables of interest (e.g., speed or direction of travel) with their associated actuators (brakes, throttle). On the left side of the figure is the controller, which is the collection of hardware (and software if it is digital) that acts on the signals according to some mathematical control law. In the digital domain, sensor signals are generally sampled at some frequency and outputs are updated at some frequency as well (usually the same frequency as the sampling frequency). Software engineers do not necessarily view the problem in this way. Their perspective is one of data flow where input data flows through a number of computational components and is then output to some device eventually. The notion of time is also perceived differently. The control engineer thinks in terms of sampling frequency, which determines signal fidelity and the design of the control laws. The software engineer is more concerned with having data processed through the system at some rate that is realized by thread scheduling during execution. These two views, which are important for analysis, can be united during system specification by defining flow specifications for various control loops. The flow-specification declaration in a component type specifies an externally visible flow through a component's ports or port groups. This flow specification can be used to capture and model the control system's frequency (often referred to as sampling time), and it provides a common characterization by which both the software engineer and the control engineer can validate the system's performance.

The above discussion points out how different domain experts view an embedded application of this type and how the AADL can address the concerns of each domain expert by the use of flow specifications. We will use these multiple views or concerns as the motivation to determine the analysis view in this example. The second step for problem analysis is to determine the analysis view, which in this case will be to determine latency values for end-to-end signal flows. Referring to Figure 2 on page 17, a number of signal flow paths and associated latencies are of concern, and those latencies are specified as part of the requirements. For example, user input via a push-button to the visual indication on the display or an engine RPM (revolutions per minute) signal to the throttle actuator would be specified in the requirements. For this specific example, we are interested in the latency of the brake pedal depression to the throttle actuator. We want a model that will allow us to specify the latency requirement, so that when the brake is applied while the cruise control is operating, the corresponding signal should reach the throttle actuator in a specified number of milliseconds. Furthermore, we want to be able to specify (from requirements) and validate (with actual execution times) the latencies of each of the individual components within the analysis. Section 3.2 describes the details of how the system components work individually and collectively, how the component and system specifications are captured in a model, and how the specifications can be validated through analysis. Section 3.5 addresses how the design of the application software architecture is performed. Performing a specification-based flow analysis within the AADL—in which the end-to-end flow is specified and then decomposed through each of the components in the system's flow path—is discussed in Section 3.4.

3.2 MODELING THE CRUISE CONTROL SYSTEM

This section builds on the structural representation capability of the AADL by describing the details of one of the application subsystems (e.g., traction control or cruise control) that were discussed in Section 3.1. The cruise control application subsystem is modeled showing the use of other AADL constructs and, at the same time, showing how state information is modeled via data variables. Later sections build on this architectural representation to model data flows and to set the basis for latency analysis by using end-to-end flows (specified and actual).

The modeling perspective is determined by deciding what question(s) must be answered through analysis and was set forth in Section 3.1.1. The fidelity of an analysis model to be generated is driven by the specific issues we want to investigate. A model should only be as detailed as necessary to answer the driving questions. This guideline will be followed throughout the development of the models that are presented in the remainder of this report. To that end, some component instantiations are exactly like their declarative part, and some may be changed depending on the desired analysis.

To develop a model with the right amount of fidelity, it is necessary to identify and understand the functionality of the components. Developing an understanding of system functionality is Step 3 of the modeling approach.

3.2.1 Understanding System Functionality

The basic functionality of each control system was discussed in Section 3.1. This section describes the details of the cruise control functionality. The purpose of the cruise control system is to maintain the speed of a vehicle, over varying terrain, when it is engaged by the driver. When the brake is applied, the system must relinquish speed control until told to resume. The vehicle must also steadily increase or decrease the current speed when directed to do so by the driver (via holding the increase speed or decrease speed buttons, or by depressing/releasing the accelerator). Releasing the buttons causes the cruise control system to control to the last speed setpoint.

Figure 4 represents the block diagram of the input-output signals for such a system. This representation has been adapted from the cruise control operation as described by Shaw [Shaw 95]. This graphical representation is typical of how functional blocks are represented in Simulink. The named signals (e.g., `cc_system_on_off`, `resume`) represent variable names that are used in code and are carried over into the AADL models as port variable names, which are described in Section 3.3.2.

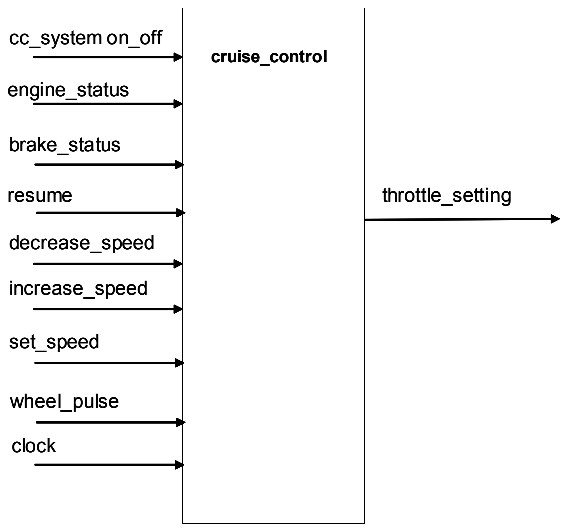


Figure 4: Cruise Control System, Simulink Input-Output Diagram

There are several inputs:

- cc_system on_off
  If on, this signal denotes that the cruise control system should acknowledge the inputs by processing them in the appropriate way.
- engine_status
  If on, this signal denotes that the car engine is turned on; the cruise control system is active only if the engine is on.
- brake_status
  This signal is asserted when the brake pedal is depressed; the cruise control system disengages when the brake pedal is depressed.
- resume
  This signal resumes the last maintained speed; it is applicable only if the cruise control system is on.
- decrease_speed
  This signal decreases the maintained speed; it is applicable only if the cruise control system is on.

- increase_speed
  This signal increases the maintained speed; it is applicable only if the cruise control system is on.
- set_speed
  When depressed, the current speed should be maintained.
- wheel_pulse
  A pulse is sent for every revolution of the wheel.
- clock
  This signal sends a timing pulse (e.g., every 10 milliseconds).

There is one output from the system:

- throttle_setting
  This signal is the positional value for the throttle setting.

Solutions to the cruise control design can be accomplished in a number of different ways, including the data-flow approach [Wang 89]; the procedure-call approach, which is based on Ward/Mellor and Boeing/Hatley structured methods techniques for modeling real-time systems [Ward 87]; the object-oriented programming approach [Booch 86, Ward 84]; the state-based approach [Smith 88]; the data structured systems-development approach [Higgins 87]; and the approach emphasizing feedback-control models [Shaw 95]. Each of these techniques represents a different approach to viewing and solving the problem. For example, in the object-oriented approach, each component is viewed as an object with its own set of methods and data scope; but in the state-based approach, the system is viewed as a series of states driven by changes of data or events. It is important to note that the AADL can model the software architecture no matter what technique is chosen. The artifacts within the AADL model are directly traceable to the implementation language or technique.

Deciding which design methodology should be used to solve real-time control problems is difficult due to the complexity of interrelated issues. Developing or evaluating the rationale for which method to use in system design is not the purpose of this report. The focus here is to show how the AADL is used once a method is chosen. To that end, we will choose the procedure-call approach. This approach uses the contents of data variables as an indication of system state. Data values are updated at some nominal rate, which is generally the conversion time of a sensor or the compute time of an algorithm. The data is then used at an appropriate time by other components. Gradually, the control values emanate from the control system and are applied to the actuator. An interesting phenomenon can occur in systems of this type. The latency of the data can vary from one computational cycle to another, and it is also affected by scheduling. The mathematics of sampled data systems in control systems analysis is based on the premise that data sampling is constant and is maintained in all of the control laws. When a control law is realized in software implementation, the timing constant can vary and can manifest itself as jitter or as seemingly unstable data on displays. The timing variation is typically introduced when the control algorithms are placed onto a processor that is running multiple tasks according to some scheduling algorithm (e.g., cyclic executive, rate monotonic, or earliest deadline first). The timing irregularities typically result from unexpected preemption of a task or a task not running because a required resource is not available The AADL can represent data that is time consistent among components and can indicate when data may be delayed (time delayed) due to scheduling side effects. This characteristic of the AADL is detailed in an earlier SEI report [Feiler 04]. In this example, as shown later, the connections between the data of two components will be represented by immediate connections. If necessary, when a delayed or previous value of the data is required, it will be represented as a delayed connection.

An abstract representation of the procedure-call perspective is shown in Figure 5. The rectangular blocks represent processing software, and the rounded-edge boxes represent data stores.

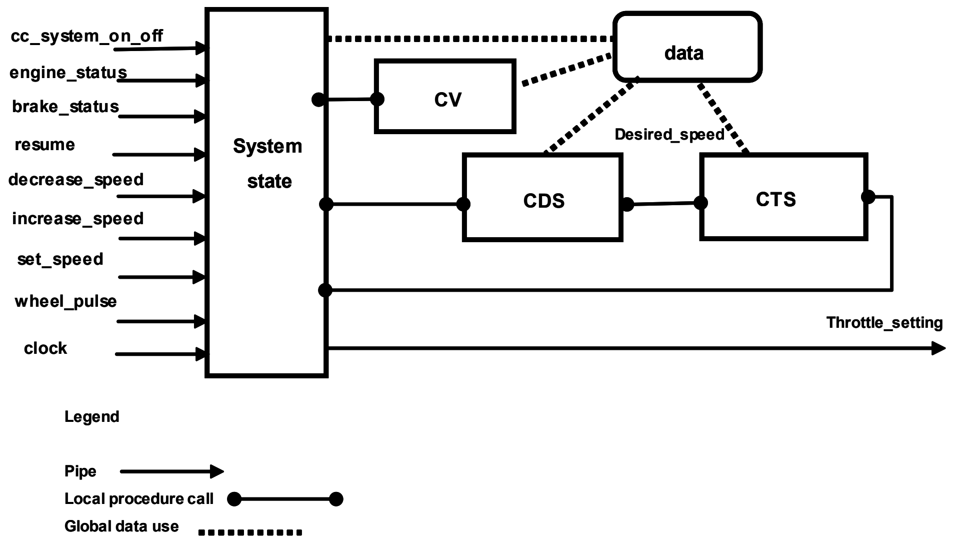


Figure 5: Representation of the Cruise Control Procedure Call

Within this cruise control implementation, a transformation from stream input to data that can be used in the procedure-call architectural style is necessary. A component called “system state” acts as the main program, scanning (sampling) the input streams and updating the global data in the shared-data component called “data.” System state also acts as the driver for the cruise control system. It periodically makes a procedure call to CalculateThrottleSetting (CTS) to calculate new throttle settings. System state periodically outputs the new throttle setting from the cruise control. This representation has a slight modification from that described by Shaw [Shaw 95], namely that the compute velocity (CV) function is explicitly represented.

CalculateThrottleSetting (CTS) and CalculateDesiredSpeed (CDS) are computation components because they do not maintain state. They provide a result based on function inputs and global data. Both CalculateThrottleSetting and CalculateDesiredSpeed read and update data via a global data use connection (shown by dotted lines) to the shared data.

Given the traction control, stability control, antilock brake, and cruise control systems, the engineering objective is to produce a safe and reliable implementation of each system’s functionality. System cost is also of concern both for the hardware and the software. Analysis of these systems could begin after composing a set of preliminary requirements. The main issue to address is what kinds of analysis should be performed. Some pertinent questions that could be asked include the following: What is the robustness of the system given a failure? Can the minimum latency specified by the control engineers be met? What is the minimum latency of a specific signal propagating in a system (e.g., application of brakes, wheel rotation)? Given the commonality of variable usage, can the system be designed to ensure predictable update order and latency of variables? These questions can be applied to each system individually or to all systems as a whole. For illustrative purposes, we will focus on the cruise control operation. The analysis for the cruise control operation can be similarly applied to the other systems (e.g., traction control or stability control).

Since the purpose of this report is to provide an introduction to using the AADL, and one of the strengths of the AADL is modeling performance-related issues, we will look at determining signal latency from sensors to actuators (specifically, the end-to-end latency of brake actuation until the throttle actuator signal goes to zero). Another issue to investigate is logical consistency of cruise control states; that issue will be addressed as an extension of the base model that will be developed.

3.3 MAPPING TO THE AADL

Having established the components and their functionality, we can begin Step 4 of the problem analysis and model development by embodying the component functionality and analysis concerns as AADL components, connections, and flows. Model development using the AADL can be done using a top-down, a bottom-up, or a combination approach. We will develop this example by first composing a top view of the system using the AADL and then decomposing the system into its constituent parts and developing the AADL declarations for each component. The software-component side of the model will then be connected by specifying the implementation. A similar approach will be taken for the hardware. The software will then be bound to the hardware to illustrate the use of component binding and composition of a system composed of both hardware and software components.

3.3.1 Representing the System Hierarchy

The top-level view of the cruise control and its input and output components is shown in Figure 6.

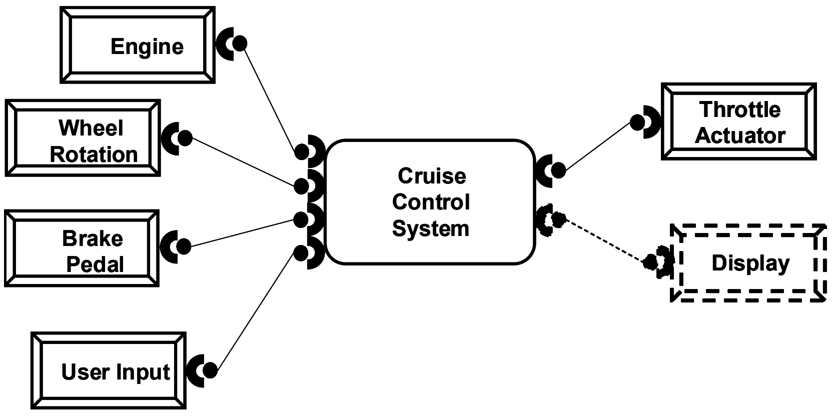


Figure 6: Context Diagram of the Cruise Control System

This figure is based on Figure 2 (on page 17), showing only the cruise control and related components. The user input device contains the cruise control on, speed up, speed down, resume, and set speed buttons. We will simplify the above example by eliminating the display functionality, which is shown as a dotted line. The component declarations of the simplified model are developed in the following section.

3.3.2 Modeling System Components

This section describes the declarative statements of the model. In the implementation section of the model, the devices are connected to the application. (The connection naming and syntax are discussed in Section 3.5.2.) In addition, this section

- discusses the abstract modeling of **`system`** components with the **`device`** construct
- shows how refinements can be made to the model via the **`property`** construct to more precisely capture key data elements that can be used in the application
- illustrates the use of the latency property to compute an end-to-end flow latency from the brake pedal, through the application software and the `throttle_actutor` **`device`**
- models the cruise control application software by using the **`system`** construct

At some point early in the modeling phase, a decision must be made regarding the modeling of events or data flow within components. In general, this decision is based on the analysis perspective used. The modeling of events and/or data may be used within the same model (i.e., latency through a component may be different based upon the modal configuration within the component).

The context diagram shown in Figure 6 is a good starting point for developing the application components. We can begin by modeling the sensors and other inputs. The AADL construct **`device`** is generally used to model components such as sensors because it provides a useful abstraction of entities that may contain complex hardware such as a CPU and associated software, allowing only the meaningful information (in this case, data and data communication) to be modeled.

For the cruise control example, the device types must be declared first. The graphical representation of the brake pedal device and associated data port is shown in Figure 7.

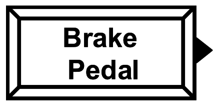


Figure 7: Brake Pedal Modeled as an AADL Device and Associated Data Port

As shown in Figure 7, the rectangular box is the symbol for an AADL **`device`**. The triangle represents the **`data port`**, which is the device's interface to the outside world. The corresponding AADL textual model is shown for all of the devices in the cruise control in the following code segment:

```
device brake_pedal
   features
      brake_status: out data port bool_type;
   flows
      Flow1: flow source brake_status {Latency => 10 Ms;};
end brake_pedal;
```

In this example, we are declaring a brake pedal device that has an output data port. The variable name of that port is `brake_status`, and the data type is Boolean. We are also designating that this device is part of a flow specification by the **`flows`** reserved word, that it is a **`flow source`**, that the variable for the flow analysis is `brake_status`, and that the name of the flow is `brake_signal_path_1`.

In order to perform latency analysis across a flow path, the latency property must be added to the flow attribute (e.g., `10 ms)`. For a **`device`**, this latency value represents the time from when the pedal is depressed to when the associated signal is available at the output **`port`**. If the **`device`** is a switch, the latency is the state change time of the switch. Since a **`device`** can represent an abstraction of something more complicated, the break pedal could contain a **`processor`** that checks for brake pedal state, performs some other processing in sequence (e.g., watchdog timer functions), and outputs the state on the `brake_status` **`port`**. The latency in this case would be the time associated with the switch state change and the computation time of the CPU. Details of flow specifications are contained in Section 3.4. If a more detailed model of the internal operation of a **`device`** is desired, the **`device`** can be replaced with a **`system`** component that would allow modeling of the hardware and software characteristics. For this example, the input/output (I/O) components details can be adequately modeled using a **`device`**. A few of the other devices are specified in a similar manner below:

```
device wheel_rotation_sensor
   features
      wheel_pulse: out data port;
end wheel_rotation_sensor;

device throttle_actuator
```

```
   features
       throttle_setting: in data port
{control_properties::actuator_voltage_range => 0.0V .. 5.0V;};
   flows
      Flow1: flow sink throttle_setting {Latency => 20 Ms;};
end throttle_actuator;
```

The **device** wheel_rotation_sensor declaration models the wheel rotation by providing an indication of wheel pulses. The data type is not declared in this particular example to show that untyped data declarations are supported. This example is also indicative of real-world modeling scenarios in which a specific data type is not known at the early stages of modeling. This declaration can be refined later in the modeling cycle.

The throttle_actuator model represents an electronic device that meters the fuel into the engine. The actuator is capable of receiving a 0−5 VDC (volts of direct current) signal that represents a relationship between input voltage and fuel output (given by the transfer function of the **device**). The **property** construct applies a limit on the minimum and maximum input values the actuator can experience. The **property** construct specifies a user-defined **property set** and **property** name. The construct must be syntactically fully qualified (e.g., control_ **properties::** actuator_voltage_range), followed by the appropriate expression for the range (e.g., 0.0V .. 5.0V;), and enclosed in braces. The **property** name must be declared in the **property set** declaration, which is discussed below.

The use of **properties** is very helpful in identifying potential mismatches of data interpretations. Setpoint data derived by the control law may manipulate data that corresponds to actual engineering units (e.g., gallons and gallons per unit time). A transformation is needed to scale the engineering units to corresponding device units (i.e., volts) before it is output. If this step is omitted, the output of the control algorithm is mismatched with the input expectations of the actuator. Using the **property** values when modeling will eliminate both data range and unit interface mismatches.

The throttle_actuator **device** illustrates the use of AADL user-defined **property sets**. **Properties** provide information about components, **features**, **modes**, **connections**, and **flows**. A **property** has a name, a type, and a value. The **property** type specifies the set of acceptable values for a **property**. A **property set** contains declarations of **property** types and **property** names that may appear in an AADL model. The AADL standard defines a set of standard **properties** (called AADL_Properties) that apply to all AADL models. A user-defined **property set** specifies additional **properties** and **property** types that must be named by qualifying them with the **property set** name. The user-defined **property set** for the cruise control example is shown in the AADL code excerpt below:

```
property set control_properties is
   dc_voltage_units: type units (mV, V=>mV*1000);
   dc_voltage: type aadlreal 0.0V..15.0V units
control_properties::dc_voltage_units;
   actuator_voltage_range: range of control_properties::dc_voltage
applies to (data port);
end control_properties;
```

In the code segment, the **property set** control_properties names three properties: (1) dc_voltage, (2) dc_voltage_units, and (3) actuator_voltage_range. The property dc_voltage_units specifies the **type** of units and the labels that are applicable in this particular setting, namely mV and V. Note that the specification of V must be written as V=> mV*1000; the syntax V=>1000*mV is not valid.

The **property** dc_voltage specifies the **type** as **aadlreal** with a numeric range as well as an associated unit. The keyword **aadlreal** is an AADL standard property type. The dc_voltage **property** declaration and the actuator_voltage_range declaration must contain a fully qualified name space of the **property** value to which it is referring. The name-space syntax is exemplified by control_properties::dc_voltage_units. The **property** name **units** is found in the control_properties **property set** and is applied to dc_voltage_units.

The actuator_voltage_range **property** uses the AADL reserved term **range of** to indicate that the dc_voltage values apply only to data ports. The reserved word **all** may be used instead of the property-owner category **data port** to indicate that the range of dc_voltages is applied to all of the AADL property-owner categories (e.g., **mode**, **port group**, **flow**, **event port**, **data port**, **server subprogram**, and **connections**). The semantics of **properties** are extensive, and the reader is referred to the AADL standard and language introduction documents [SAE 04, Feiler 06a] for complete coverage of this topic.

Added to this **device** model is a **property** of the wheel_pulse variable to specify **units** (e.g., PPS [pulses per second]) and the minimum and maximum pulse rate (e.g., 0-1000 PPS). This **property** illustrates that the **property** construct can clarify some sensor characteristics that may be specific to the application or manufacturer. For example, multiple vendors may provide wheel rotation sensors with different characteristics. The minimum and maximum rates specified could be used to distinguish what sensor is used in this particular modeling application.

Another reason to explicitly state some sensor characteristics when modeling the sensor is to provide limit-checking information that can be used by the application. For example, if the actual value exceeds the known physical limitation of a device, a check could be performed in the model (and presumably reflected in the actual code), and, if the limit were exceeded, some fault-tolerant scheme could be invoked.

The throttle_actuator device also contains a flow specification statement, indicating that it is a **flow sink** and that the **device** has latency of 20 ms. Given that the throttle_actuator **device** is an abstraction of the actual throttle actuator, the mechanism to carryout the throttle signal could be simple or complex (similar to the brake_pedal abstraction). The 20 ms value represents the time from when the signal is applied to the input port to when the output is completed.

The remaining set of devices—engine, resume button, speed up button, speed down button, and speed set button—are modeled in a similar manner and are documented in Appendix C beginning on page 69.

In addition to identifying the external devices, we also must define the application's software component (in this example, the cruise control). Software components can be hierarchical, in that general functionality can be decomposed into smaller functional subsystems. The AADL supports this hierarchical relationship for many of the modeling objects (e.g., **`system`**, **`mode`**, **`port`**, **`port group`**, and **`processor`**). We use the **`system`** construct as a generic application software component to represent some software functionality, the interface to the external world (via ports) and information or event flow (via flow statements) from an input port to an output port. Design decisions about thread realizations, concurrent execution, and space partitioning can be deferred until later in the design process and can be contained within the same model. Flow latency analysis can be performed with this high-level model, while detailed scheduling can be performed later when **`thread`** and **`processor`** details are added to the model. Adding the **`thread`** information allows scheduling to be performed by a scheduling analysis plug-in.

Figure 8 shows the AADL graphical representation of a named system with a named in data port and a named out data port. The **`system`** construct was used previously in Section 3.1 to model each of the four major control applications in an automobile.

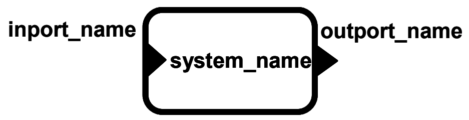


Figure 8: AADL System Graphic Symbol with an In Port and Out Port

System component interfaces are characterized by their features, flow specifications, and properties. This section focuses on ports as features. Ports have names and a direction of flow. In the case of data ports, they may also have a data type. The port name identifies the port; the keywords **`in`**, **`out`**, or **`in out`** define the direction. The keyword **`in`** specifies that data flows into the system, the keyword **`out`** indicates that data flows out of the system, and the keyword **`in out`** specifies that the data is bidirectional.

We use a top-down decomposition approach to develop the cruise control application. At the top level, the cruise control application can be modeled as a **`system`** component. The **`system`** is graphically represented as shown in Figure 9.

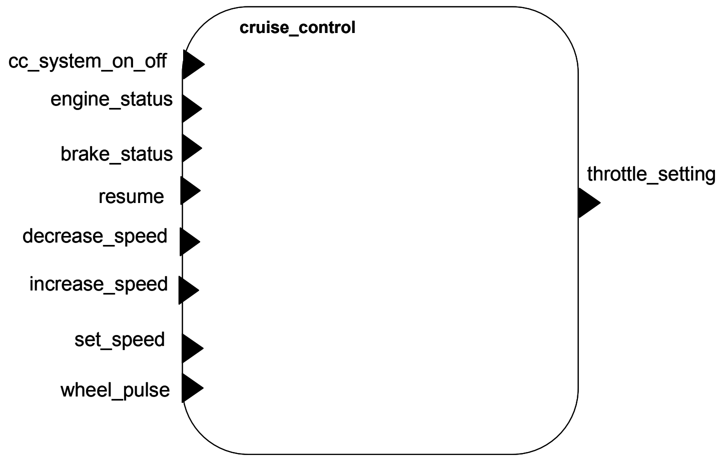


Figure 9: Cruise Control Represented as an AADL System

The associated textual declaration of the cruise control **system** component is provided below:

```
system cruise_control
   features
      cc_system_on_off: in data port;
      engine_status: in data port;
      brake_status: in data port;
      resume: in data port;
      decrease_speed: in data port;
      increase_speed: in data port;
      set_speed: in data port;
      wheel_pulse: in data port;
      throttle_setting: out data port;
   flows
      brake_flow_1: flow path brake_status -> throttle_setting;
end cruise_control;
```

In this example, all ports are represented as data ports. They support unqueued communication of state data. Data from the sensors flowing to other components can be viewed as a signal stream. This communication choice has several important characteristics. The stream rate among different sensors may be different, which means that the port sampling rate must be fast enough so a change of state will not be missed. Decisions must be made later in the analysis to define what the rate should be and to determine if it makes sense to group sensors with similar rate characteristics to possibly achieve better performance. A related data communication issue is that of sending delta information instead of state information for each sample. The wheel sensor could send an accumulated pulse count, or it could send a delta pulse count since the last communication time. This tradeoff affects the communication scheme to be used. If a delta value is lost (e.g., the port is not read at the appropriate time), the next delta data value is communicated and used in the computation. In computing velocity, this lost value may be masked due to the inertia of the system and the net effect will be unobservable. However, if the data is used to compute distance, the accumulated distance would be noticeably incorrect.

In this example, the ports were named identically to the data that is expected on each port (i.e., **`port`** named `brake_status` is to receive **`data`** named `brake_status`, of **`type`** specified by the **`device`** `brake_pedal` declaration).

Some observations about **`data port`** specifications are worth noting. Looking at the `brake_pedal` **`device`** specification (Figure 7), note that the **`device`** declaration specifies an **`out data port`** of **`type`** Boolean. The intent is to connect this **`out data port`** to the corresponding input port of the cruise control application. Doing this will ensure a data matching between the **`data source`** and the **`data sink`**. Looking at the cruise control application declaration discussed above, note that the `brake_status` **`in data port`** does not have a data type specified. The AADL states that interfaces must be completely specified, including data type. It is acceptable not to include the data type specification in the declaration. In this case, the intention is to use the declaration as a template, and the data type would be specified in the implementation.

When comparing Figure 9 to Figure 5 (see page 27), we see that the clock signal is not shown. We did not show the clock because the intent of this exercise is to represent the data-flow view of the system to illustrate flow-path modeling, which can be used to perform subsequent latency computations. The amount of detail represented thus far is sufficient to meet the intended analysis. The clock represents the sampling frequency of the system. To drive the model to more detail, one would model a thread (e.g., `sample_thread_10hz`) with a periodic dispatch protocol and with a dispatch frequency of `10 ms`.

Alternatively, one could model the clock as a **`device`** (e.g., `clock_10hz`) with a `Device_Dispatch_Protocol` **`property`** specified with the appropriate period. The clock **`device`** would also contain an **`out event port`** connected to the `cruise_control_system` (`cc_app.impl`) via a corresponding **`in event port`** that would trigger a processing **`thread`** within the application. Refer to the AADL standard, Sections 5.3 (Threads), 6.4 (Devices), and 8.1 (Ports) for details regarding event utilization and execution properties [SAE 04].

The cruise control **`system`** declaration also contains a flow path specification indicating that a named **`flow path`**, `brake_flow1`, begins at the input **`port`** `brake_status` and ends at the **`port`** `throttle_setting`. The declared **`flow path`** name will be used again to model the cruise control implementation. Section 3.3.3 describes the individual software application components contained in cruise control and their flow and latency values. Section 3.4 describes the construction of the cruise control implementation using those components.

3.3.3 Identification and Modeling of Application Components

The top-level view of the cruise control application can now be refined into the computational subcomponents that make up the details of the cruise control. Recall that the architectural approach taken in this design is that of procedure calls. As denoted in Figure 5 (on page 27), the main functional components to be modeled as subprograms are

- `in_control`: Acquire the inputs and perform state checking.
- `compute_velocity`: Compute the velocity from wheel pulses.
- `compute_desired_speed`: Compute the speed based on a control law.
- `compute_throttle_setting`: Convert the speed to throttle position.

Each software component is represented in Figure 10 using the AADL graphical notation. Each software component is shown as a system, configured with in ports and out ports. The corresponding data name is adjacent to each port.

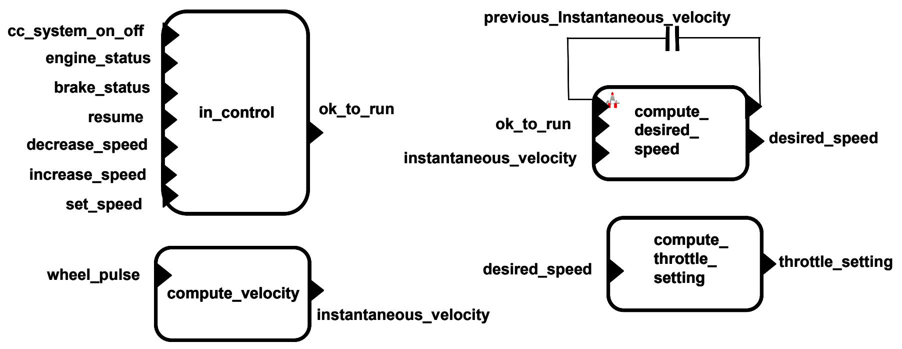


Figure 10: Cruise Control Software Components Depicted as AADL System Components in AADL Graphic Notation

We have abstracted away the details of the functionality and have shown them as computational entities with input and output ports. The AADL textual type declaration for each of these components is shown below:

```
-- the in_control software component
system in_control
   features
      cc_system_on_off: in data port;
      brake_status: in data port bool_type;
      resume: in data port;
      decrease_speed: in data port;
      increase_speed: in data port;
      set_speed: in data port;
      engine_status: in data port;
      ok_to_run: out data port;
   flows
      FS1: flow path brake_status -> ok_to_run; {Latency => 30 Ms;};
end in_control;
```

```
-- the compute velocity software component
system compute_velocity
   features
      wheel_pulse: in data port;
      instantaneous_velocity: out data port;
   flows
      FS1: flow path wheel_pulse -> instantaneous_velocity;
end compute_velocity;

-- the compute desired speed software component
system compute_desired_speed
   features
      ok_to_run: in data port;
      instantaneous_velocity: in data port;
      current_instantaneous_velocity: out data port;
      previous_instantaneous_velocity: in data port;
      desired_speed: out data port;
   flows
      FS1: flow path  ok_to_run -> desired_speed {Latency => 40 Ms;};
      FS2: flow path instantaneous_velocity -> desired_speed;
end compute_desired_speed;

-- the compute throttle setting software component
system compute_throttle_setting
   features
      desired_speed: in data port;
      throttle_setting: out data port;
   flows
      FS1: flow path desired_speed -> throttle_setting {Latency => 50
Ms;};
end compute_throttle_setting;
```

Each cruise control software component has been declared as shown in the preceding code segment. In the AADL syntax of the `in_control` software component, the **`system`** reserved word denotes the beginning of the component's description; the reserved word **`end`** marks the end of that description. The **`features`** of this component are named input or output ports (e.g., `brake_status:` **`in data port`**). The general format of this feature declaration is `port_name: port_direction port_type`. The port name is a necessary label used to designate the source and destinations of connections (see Section 3.5.2). Port direction specifies the direction of data flow. The data flow can be in, out, or in out (i.e., there can be incoming and outgoing communication). In order to compute the latency of a specified **`flow path`**, each application component declaration that composes the **`flow path`** of interest includes a flow specification and the associated computational latency. This latency is specified as shown in the preceding code segment, using the `latency` property of the AADL. The specification of data flow is used in computing end-to-end latency of specific data of interest (see Section 3.4.1).

The entire cruise control application has been abstracted to encompass the basic computation of the software components and their interfaces to other components via data ports. This abstraction represents a complete declarative model of the cruise control application and its associated devices. The complete model is in Appendix C (beginning on page 69).

3.4 FLOW ANALYSIS

As discussed earlier in this report, the AADL contains support for modeling data and control flows as part of architectural descriptions. A flow specifies the flow of data or events through multiple components along a sequence of components and connections. A component—such as a thread, process, or system—has a flow specification as part of its component type declaration (i.e., an externally visible specification of a flow of data or control from a component's in ports to its out ports). The purpose of providing flow specifications is to support many forms of end-to-end analysis completely through a system or within a subset of components. These end-to-end analyses include end-to-end timing and latency, numerical error propagation, processing sequences of domain objects, and quality-of-service resource management based on operational modes. To support such analysis of software components using the AADL, we must specify relevant properties such as ports, flow specifications, and flow-specific properties. For example, a flow-specific property could be the expected maximum latency that the data within a component would experience, as well as the actual latency. The expected maximum flow-specific property, which presumably would be determined during design, could be checked against the actual implementation (whose results would be contained in the actual latency) to see if the design assumption was met.

An explicit flow-path declaration indicates that input sent to a component via its **`in port`** will emerge on its **`out port`**. This is the case when a component processes input from an **`in port`** and makes the results available as different data on different **`out`** ports. Similarly, a component may merge data from different **`in`** ports into a single resulting data element available on an **`out port`**. Multiple flow specifications can be defined involving the same port. For example, data coming in through an **`in port`** is processed and derived data is sent out through two or more different **`out`** ports. Naming of flow specifications allows multiple **`flows`** to be specified through the same component, in particular, multiple **`flows`** through the same **`port`**.

A flow may start within a component (making the component a flow source), and it may end within the same component (making the component a flow sink). The flow source and flow sink are also specified for a flow that spans a number of components. A flow-source specification is indicated with a **`flow source`** reserved term, while a flow-sink specification is indicated with a **`flow sink`** reserved word. A flow from an in port to an out port is referred to as a **`flow path`**.

3.4.1 Flow Specifications

End-to-end flows are composed of three entities: a flow specification, a flow declaration, and a flow implementation. A flow-specification declaration indicates that information logically flows from one of its incoming ports, parameters, or port groups to one of its outgoing ports, parameters, or port groups. In order to analyze a system for its flow, a flow specification must be written for the top-most component over which the flow is to be determined. The specification can be written for any component within the component hierarchy.

For the cruise control system, we want to determine the latency from the `brake_status` input port to the `throttle_setting` output port. The flow of interest is shown in Figure 11.

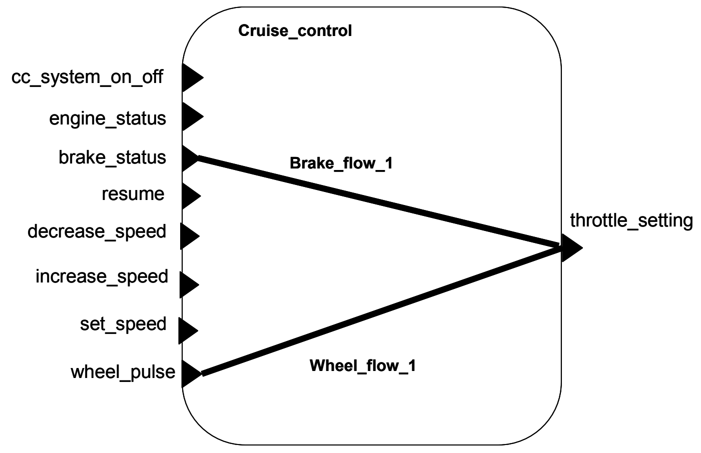


Figure 11: Top-Level Flow Specification Using AADL Graphical Notation, Showing Brake and Wheel Flow Paths

Flow specifications must be named; therefore the flow of interest is Brake_flow_1. There is another flow shown named Wheel_flow_1. A flow-specification declaration in a component type specifies an externally visible flow through a component's ports, port groups, or parameters. The flow can be through a component (a **`flow path`**), a flow-originating component (**`flow source`**), or a flow-ending component (**`flow sink`**). The syntax of the flow specification for the cruise control application is shown below:

```
flows
   brake_flow_1: flow path brake_status -> throttle_setting;
```

The general case is of the following form:

```
flows
      flow_path_name: flow path source_point -> sink_point;
```

where `flow_path_name` maps to `brake_flow_1` (i.e., the name of the flow path), **`flow path`** is the AADL reserved term, `source_point` maps to `brake_status` (the input port name), and `sink_point` maps to `throttle_setting` (the output port name). The flow statement is contained within the system declaration of the component.

In the AADL, the overall flow must have a declared beginning and end points. The beginning and end points must be in declarations in the following form:

```
flows
      flow_specification_name: flow source flow_source_spec;

flows
      flow_specification_name: flow sink flow_sink_spec;
```

For the cruise control application, the brake-pedal device and the throttle-actuator device are the desired source and sink of the flow analysis. The flow source and sink statements appear in the device declarations (Section 3.3.2) as follows:

```
flows
      Flow1: flow source brake_status

flows
      Flow1: flow sink throttle_setting;
```

Some things to note in the semantics are that the flow specification name is `Flow1`, the flow-source specification is the brake-pedal **`data port`** named `brake_status`, and the throttle actuator **`data port`** is named `throttle_setting`.

As mentioned earlier, specification-based flow analysis is possible at this level of abstraction. Properties can be associated with a flow specification. The allowed properties include expected latency, actual latency, expected throughput, and actual throughput. For the brake-flow specification above, the syntax to include an expected latency value would take the following form:

```
flows
   brake_flow_1: flow path brake_status -> throttle_setting;
properties
   Expected latency: 200ms;
```

Specifying the latency at this level clearly conveys the design goal of the system. Each of the declared subsystems (such as CalculateDesiredSpeed [CDS]) would have an associated latency property in its system declaration. A check could be performed to ensure that the overall flow latency is equal to or greater than the summation of each subsystem's expected latency property. In **`system`** implementations, each component would have an actual latency property specified, and this latency property would hold the actual execution time of the subsystem as measured by the appropriate analysis tool or plug-in. The sum of the actual values would be compared to the overall specification to see if the latency requirement was met. A comparison of actual versus expected latency on a subsystem-by-subsystem basis could also be performed to identify any component that exceeds its expected value.

3.4.2 Flow Implementation

A flow specification specifies an externally visible flow through a component's ports, parameters, or port groups. A flow-implementation declaration in a component implementation specifies how a flow specification is realized in the implementation: as a sequence of flows through subsystems (i.e., subcomponents) along connections from the flow-specification in port to the flow-specification out port. Since flow is realized when code is executed, processes and threads must be considered.

Figure 12 shows a graphical notation of the system implementation for the brake-flow path, F2. It shows the cruise control implementation (named cruise_control.impl) and contains the cruise control process. Connection names are assigned to the connections linking the port names at the system level to their logical input to the process. For example, the brake-status port is connected to the cruise control process with connector C2.

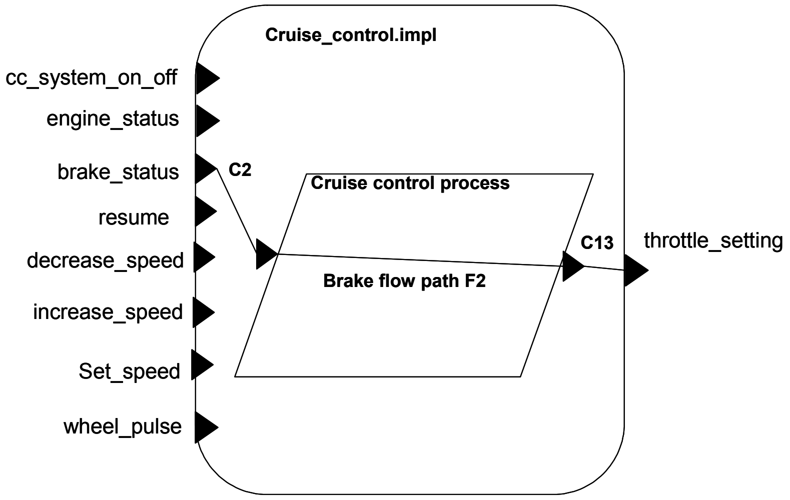


Figure 12: AADL Graphical Notation of a Flow Path Within a Single Thread in a Process

Connection C13 connects the output variable `throttle_setting` to an external interface. A connection will have to be made to connect the variable to the throttle actuator. The throttle actuator will then receive the data from the port named `throttle_setting`. The flow-path implementation would be written as `F2: brake_status->C2->cruise control process.brake` **`flow path`** `F2-> C13->throttle setting`. The implementation in the AADL syntax is discussed in Section 3.5; the flows through the subcomponents are discussed in Section 3.5.2; and the end-to-end flow syntax and analysis view is discussed in Section 3.5.5.

Systems with multiple flows can be modeled as shown in Figure 13. This figure depicts a system implementation, my_system.impl, containing two processes, P1 and P2, that contribute to implementing a flow specification, F1 (not shown in Figure 13). The flow path, F1, is realized in the implementation section of the model, and would be written as `F1: port_1->C1->p1.F5->C2->p2.F7->C3->port_3`. The implementation details for our flow example is described in Section 3.5.5

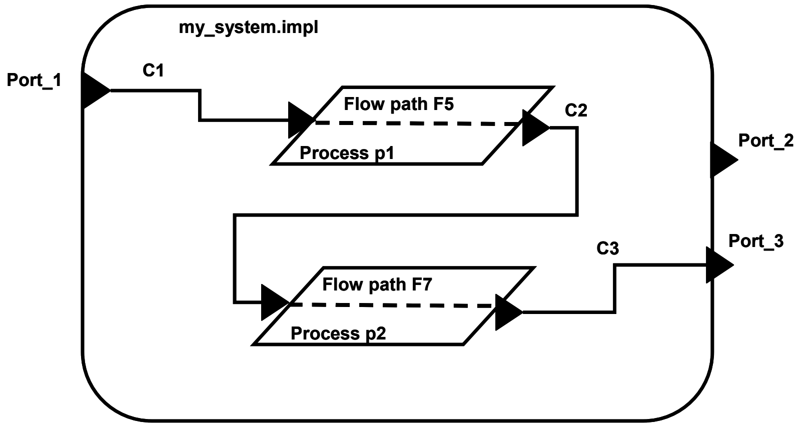


Figure 13: Generic Representation of a Flow Implementation through Two Processes

3.5 DEVELOPING THE SYSTEM IMPLEMENTATION

At this point in the report, a top-level model of the cruise control application has been developed. By subsequently decomposing it into its subsystems and forming declarative models of those systems, we have illustrated the hierarchical modeling approach using the AADL. Connections among the systems have been shown graphically to illustrate data-flow intent, but modeling of those connections has not been performed. System flows have been defined, and a specification-based analysis approach has been discussed. In this section, we complete the model by showing how binding to hardware occurs and, subsequently, how system implementations are modeled based on the declarations, how connections are made for a specific implementation, and how flow implementations can be specified to illustrate a complete flow analysis.

3.5.1 Binding to a Computing Platform

A powerful characteristic of the AADL is its ability to model hardware components of the target system. Binding the software components to the associated hardware components allows the modeler to specify and evaluate the interactive effects of the complete system. For example, evaluating system software on a uniprocessor system versus a distributed parallel processor system could provide speed-up numbers for comparative evaluation. Another advantage of this modeling approach is the ability to test and evaluate the system for problems that occur due to concurrency (e.g., access order to variables, deadlocks) that could surface and are generally not exposed until actual integration and testing are performed.

Specifying a computing processor and associated components can generally be done anytime during the modeling effort. Using the AADL, software applications can be mapped onto any number of processors. Depending on the type of the analysis, processors do not have to be declared or implemented. Since we are performing a flow analysis, a processor type is specified. Doing this can help ensure that, later in the actual implementation, the execution time will be consistent with the specified target processor that is acquired.

Multiple processors may also be specified and the application distributed among them. This is often done to ensure that the desired overall computation time meets the specification. The cruise control example is small enough to fit on a single processor, and that is how it is implemented. As more automotive subsystems are modeled, it may be necessary to investigate their application to a uniprocessor or distributed processors to decrease overall computation time.

As in the case of software-component modeling, different system views can be developed and evaluated. The scope and the degree of the abstraction for the hardware-runtime platform are dictated by the amount of detail desired for analysis. At a very general level, declaring the processor, memory, and bus with minimal characteristics is sufficient. Only the features that would support the analysis would have to be declared. For example, specifying the total amount of physical memory and the amount in each partition would be sufficient to check if each software subsystem would fit into each partition (i.e., program footprint budget). Similarly, memory speed could be declared, which would aid in analyzing actual runtime performance.

Once a processor and memory have been declared, they must be connected by a bus. Declaring a bus type of, for example, PCI (Peripheral Component Interconnect) or VME (Virtual Machine Environment) encompasses the protocol and speed characteristics defined by each of those standards. Custom bus configuration can also be developed and modeled. One example is a "PCI-2" bus that would embody the current PCI protocol but with increased speed. This type of modeling could be used to evaluate end-to-end latency with systems using input/output (I/O) devices that are attached to either the slower or faster bus.

For the cruise control example, a relatively simple hardware platform will be declared, namely a uniprocessor system composed of a Pentium class CPU, a PCI bus, and 256 megabytes of Synchronous Dynamic Random Access Memory (SDRAM), which is typical of many embedded system applications.

The general syntax for a processor declaration is shown below:

```
processor processor_name
   features
      controller_cpu: requires bus access bus_name;
end processor_name;
```

The `processor_name` can be any user-defined name reflecting the target processor. The features section of the processor declaration may contain references to a required bus access, server subprogram, or port. A **`processor`** component must contain at least one **`memory`** component and must require at least one **`bus`** access. The **`bus`** access is specified in the declaration (`bus_name`), whereas the **`memory`** is specified within the implementation and is not shown in the declaration template above.

A number of standard properties can be specified within an implementation (or within a declaration). These include `source_language`, `source_code_size`, `source_data_size`, `thread_limit`, and `scheduling_protocol`. The user can make the **`processor`** model as detailed as necessary. Deciding whether or not to specify the **`properties`** for the component type or implementation depends on the modeler's decision of what is the basic set of **`properties`** that each component will need as part of a base configuration.

For the cruise control example, the **`processor`** declaration is shown below:

```
processor PENTIUM
   features
      controller_cpu: requires bus access PC104_ISA_16BIT;
end PENTIUM;
```

The only necessary specification is **`requires bus access`**.

The general syntax for the **`bus`** component declaration is provided below:

```
bus bus_name;
   features
end bus_name;
```

The bus-declaration features can contain a **requires bus access** clause and flow specification. The **bus** implementation can contain a number of predeclared AADL **properties** that include `allowed_connection_protocol`, `allowed_access_protocol`, and `allowed_message_size`. These **properties** are useful for ensuring consistency among entities that would connect to the bus. For example, devices that communicate via TCP/IP could be connected only to buses that support TCP/IP.

The declaration of the **bus** for the cruise control is shown below:

```
-- hardware platform declaration
bus PC104_ISA_16BIT
end PC104_ISA_16BIT;
```

The **memory** declaration below shows how the **bus** declaration is used. The **bus** that the **memory** connects to is the 16-bit Industry Standard Architecture (ISA) standard bus, contained within a PC104 form factor.

```
memory SDRAM
   features
      controller_memory: requires bus access PC104_ISA_16BIT;
end SDRAM;
```

To complete the hardware-component declarations, a **system** component must be declared that will eventually hold the hardware components. This declaration is accomplished by the following AADL syntax:

```
system cc_computer
end cc_computer;
```

This declaration is not quite correct as it stands. Recall that we have a number of devices that must be connected to the computer. The AADL states that, in order to use devices, the executing processor must be connected to a device via a bus. To effect this connection, a statement must be added to this declaration indicating that the system provides bus access. This connection is accomplished in the following example:

```
system cc_computer
-- a declaration for cc_computer to be composed of processor, memory,
-- and bus in its implementation
-- Needs to provide bus access so the devices in cc_application can
-- communicate with cpu
-- Devices need to be attached to a bus.
   features
      device_bus: provides bus access PC104_ISA_16BIT;
end cc_computer;
```

Since we are assembling the software components of the cruise control into a system, we must also declare a component that contains those components. This is accomplished by the following declaration:

```
system cc_app
end cc_app;
```

This declaration is not quite correct either for the same reason discussed above for the `cc_computer` component. The only difference in this declaration is that we must use the **`requires bus access`** reserved term, as shown below.

```
system cc_app
   features
      device_bus: requires bus access PC104_ISA_16BIT;
end cc_app;
```

This declaration indicates that the `cc_app` system requires access to a bus of type `PC104_ISA_16BIT`. The resolution of the `device_bus` named in each of the declarations will be addressed in Section 3.5.3.

3.5.2 Component Connections

At this point, all of the components necessary for analyzing the model for end-to-end latency have been declared. The components now have to be connected to provide the data-flow paths. A connection is a linkage that represents communication of data and control between components. The AADL supports three types of connections: port connections, parameter connections, and access connections.

- Port connections represent the transfer of data and control between two concurrently executing components (i.e., between two threads or between a thread and a processor or device).
- Parameter connections represent the flow of data through the parameters of a sequence of subprogram calls (i.e., between units of sequential execution within a thread).
- Access connections represent access to shared data components by concurrently executing threads or by subprograms executing within a thread. They also represent communications between processors, memory, and devices by accessing a shared bus.

Parameter connections represent the flow of data between the parameters of a sequence of subprogram calls within a thread of execution. Parameter connections may be declared as follows:

- from an **`in data`** or **`event data port`**, or an **`in out data`** or **`event data port`** of the **`thread`** containing a **`subprogram`** (or subcomponent) call
- from an **`in parameter`** or **`in out parameter`** of the containing **`subprogram`** to a **`subprogram`** call's **`in`** or **`in out parameter`**
- from a **`subprogram`** call's **`out`** or **`in out parameter`** to an **`out`** or **`in out parameter`** of the containing **`subprogram`** call's **`in`** or **`in out parameter`** of the containing thread's **`out`**, **`in out data`**, or **`event data port`**

In other words, the parameter-connection declarations follow the containment hierarchy of subprogram calls inserted in other subprograms.

For parameter connections, data transfer occurs at the time of the subprogram call and call return. In the case of subprogram calls to a server subprogram in other threads, the data is first transferred to a local proxy and from there passed to the remote subprogram.

The rules for component connections are that (1) **`out`** ports must be connected to **`in`** ports (or **`in out`** ports) and they must be of the same **`type`** and that (2) components connected at the same level of the hierarchy must follow the hierarchy. The architecture of the fully connected system is shown graphically in Figure 14.

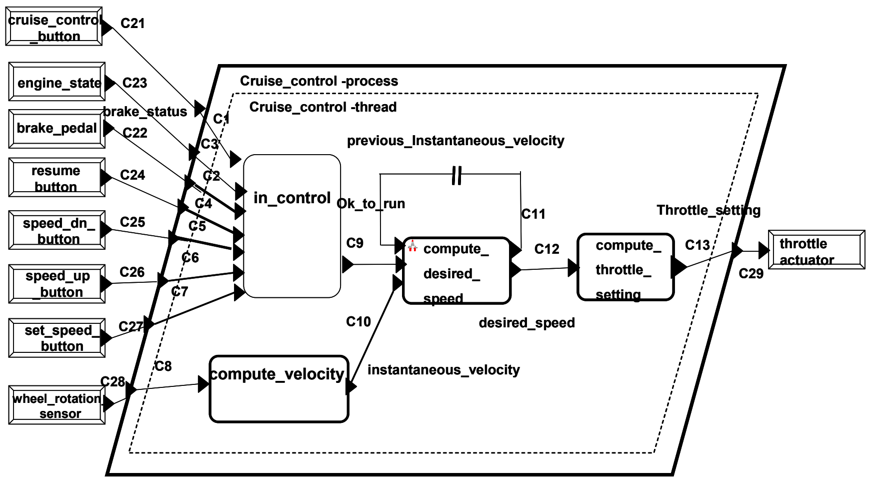


Figure 14: An Application Software Component View of the Cruise Control Showing All the Device and Component Connections.

Assembling the system graphically helps when writing the textual representation. The assembly (or connection) of the components can occur only in an implementation of the declared components. The AADL **`system implementation`** reserved term is used to compose the system instance. The general syntax is shown below:

```
system implementation  implementation_name.impl
   subcomponents
      subcomponent_identifier_1: system subcomponent_name_1;
      subcomponent_identifier_2: system subcomponent_name_2;


:
      subcomponent_identifier_n: system subcomponent_name_n;

   connections
connection_identifier_1:  data port port_name ->
component_name_1.port_name;
connection_identifier_2:  data port port_name ->
component_name_2.port_name;
:
:
```

```
connection_identifier_n:  data port port_name ->
component_name_n.port_name;


   flows
flow_path_name: flow path flow_source_name -> connection_1-
>component_1.FS1->connection_2->component_2.FS1-> connection_3-
>component_3.FS1->connection_4->flow_sink_name;

end implementation_name.impl;
```

The **`system implementation`** lists all of the **`subcomponents`**, **`connections`**, and **`flows`** (if necessary) that are to be contained for a specific implementation. Within the implementation, each applicable component previously declared for the analysis must be listed (`subcomponent_name`) and named with an implementation name (`subcomponent_identifier`).

The data-flow connections are implemented within the **`connections`** section. Connections are named to provide a reference (`connection_identifier`). The connections' implementations specify the path of data from a specified port (`port_name`) to the next entity contained in the path. This entity is specified by identifying the component name and its associated port name (`.port_name`).

Since a flow analysis of this example is desired, the flow section must be included within the implementation. The flow implementation is named `flow_path_name`, which is a unique name for this particular implementation. The flow path contains an enumerated string that begins with the source; it continues with the named connections, the next component within the path, and the component's associated incoming port name into which the data flows; and it ends with the flow sink. This flow path specifies in an unambiguous way the entire data path for analysis. If a system contains other systems (e.g., in a hierarchical fashion), each of them must have a flow specification, and each connection and port through each level traversed by the flow must be enumerated in the implementation.

For this specific example, the system implementation for the cruise control application is as follows:

```
system implementation  cruise_control.impl
-- List the declared components that comprise this implementation
   subcomponents
      I_C: system in_control;
      C_V: system compute_velocity;
      C_D_S: system compute_desired_speed;
      C_T_S: system compute_throttle_setting;
-- make the connections among the components as desired
      C1:  data port cc_system_on_off -> I_C.cc_system_on_off;
      C2:  data port brake_status -> I_C.brake_status;
      C3:  data port engine_status -> I_C.engine_status;
      C4:  data port resume -> I_C.resume;
      C5:  data port decrease_speed -> I_C.decrease_speed;
      C6:  data port increase_speed -> I_C.increase_speed;
      C7:  data port set_speed -> I_C.set_speed;
      C8:  data port wheel_pulse -> C_V.wheel_pulse;
```

```
        C9:  data port I_C.ok_to_run -> C_D_S.ok_to_run;
        C10: data port C_V.instantaneous_velocity ->
C_D_S.instantaneous_velocity;
        C11:  data port C_D_S.current_instantaneous_velocity ->
C_D_S.previous_instantaneous_velocity;
        C12:  data port C_D_S.desired_speed ->  C_T_S.desired_speed;
        C13:  data port C_T_S.throttle_setting -> throttle_setting;
-- the flow, beginning at the input port, through each of the
necessary
-- subcomponents, to the out port
   flows
        brake_flow_1: flow path brake_status -> C2->I_C.FS1->C9-
>C_D_S.FS1-> C12->C_T_S.FS1->C13->throttle_actuator;

end cruise_control.impl;
```

The cruise control application has been given the extension ".impl" as a name for the specific implementation. The system components that have previously been declared and that are included in this specific instantiation are listed in the subcomponent section and are named. Other configurations may be made by composing implementations with different names (i.e., `cruise_control.multi-threaded`) and with a different set of declared components and connections.

3.5.3 Integrating the Application Software and Hardware

At this point in the example, the software components and hardware components have been declared, and the implementation of the application has been developed (`cruise_control.impl`). Pictorially, the complete system hierarchy is shown in Figure 15.

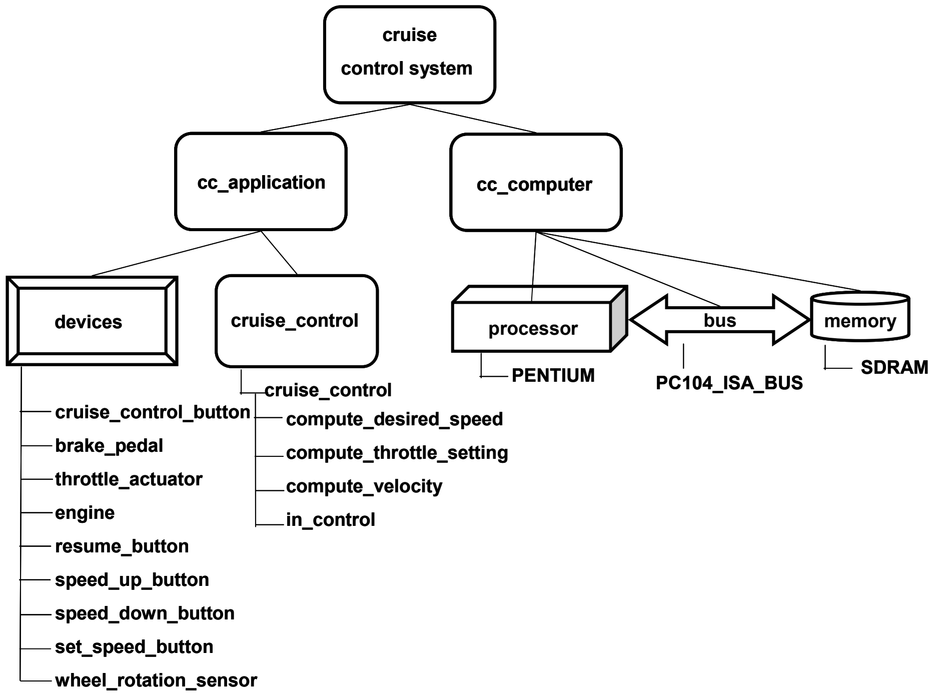


Figure 15: Complete System Hierarchy for the Cruise Control Showing Software Application Components and Associated Hardware Components

The left side of Figure 15 shows the devices and system components that have been declared and, in this particular case, constitute the cc_application system. The right side of the figure shows the hardware components that have been declared and that constitute cc_computer. To compose the cc_computer system, the hardware implementation must be specified. The general declaration of cc_computer, which was developed in Section 3.5.1, is used as the basis of the implementation. The hardware implementation will be named for a specific (but fictitious) automobile manufacturer (in this case, CompanyZ), and the AADL representation is shown below:

```
system implementation cc_computer.CompanyZ
 subcomponents
 CompanyZ_memory: memory SDRAM;
 CompanyZ_bus: bus PC104_ISA_16BIT;
 CompanyZ_processor: processor PENTIUM;
 connections
 bus access CompanyZ_bus -> CompanyZ_memory.controller_memory;
 bus access CompanyZ_bus -> CompanyZ_processor.controller_cpu;
end cc_computer.CompanyZ;
```

The implementation consists of the memory, bus, and processor subcomponents and the connections between the bus access to the memory and processor. This example can be used as a general template for a uniprocessor system.

To complete the model of the entire system (application and hardware platform), the cruise control system (the top system icon in Figure 15) must be declared and implemented. This declaration and implementation is accomplished in the following AADL model:

```
system CompanyZ_cruise_control_system
-- a declaration for Company Z cruise control system to be composed
-- of computer + applications sw in its implementation
end CompanyZ_cruise_control_system;
-- Composing the complete system
system implementation CompanyZ_cruise_control_system.impl
   subcomponents
      CompanyZ_computer: system cc_computer.CompanyZ;
             CompanyZ_software: system cc_app.impl;
   connections
      C1: bus access CompanyZ_computer.device_bus ->
CompanyZ_software.device_bus;
end CompanyZ_cruise_control_system.impl;
```

The system `CompanyZ_cruise_control_system` is declared. The declaration is left empty, indicating that no external interfaces are present. The implementation of the complete system is modeled using the **`system implementation`** reserved term. This specific implementation is given a name (`CompanyZ_cruise_control_system.impl`), and the component implementations of the pieces to be used are enumerated using the **`subcomponents`** reserved word. The component implementations for this system are `cc_computer.CompanyZ` and `cc_app.impl`.

3.5.4 Connecting the Devices to the Bus

To complete the cruise control model, the devices must be connected to the CPU-memory bus. The AADL specifies that the executing processor that has access to the device must be connected to the device via a bus. The declarations of `cc_app` and `cc_computer` were discussed in Section 3.5.1, which cited the use of the following AADL reserved words: **`requires bus access`** and **`provides bus access`**. These declarations, in essence, named an external interface called `device_bus,` in which access to the CPU-memory bus (PC104_ISA_16BIT) is made available. The `cc_app` implementation, which inherits access to the `device_bus` via the `cc_app` declaration, must connect the devices configured in the `cc_app` implementation to the hardware implementation. This connection is accomplished by using the AADL **`connections`** reserved word in the system implementation of `CompanyZ_cruise_control_system.impl` discussed in Section 3.5.3. The connection is named C1, and using the reserved word **`bus access`** indicates that the devices attached to the named `device_bus` of the software are connected to the computer. This **`connections`** subclause resolves the `device_bus` name that was specified in a specific implementation of `CompanyZ_computer` which is composed of the `cc_computer.CompanyZ`. Similarly, the CompanyZ.software is composed of the `CC_app.CompanyZ`.

3.5.5 Specifying the End-to-End Flow for Analysis

We now have in place sufficient information to specify an end-to-end flow path. To briefly recap:

- devices and application software components have been declared (Sections 3.3.2, 3.3.3)
- individual device and component flow latencies have been defined (Sections 3.4, 3.4.1)
- flow implementation across the application software components has been specified (Section 3.4.2)
- component bindings to a computing system have been performed (Section 3.5.1)
- connections of all the components have been constructed for the complete system implementation (Section 3.5.2)

Having connected all of the software application components and devices allows us to specify end-to-end flow paths that are of particular interest for analysis. In this example, we want to specify the time it takes for a signal originating at the brake pedal to arrive at and move the throttle actuator.

The end-to-end flow specification can be viewed as the maximum latency allowed for this path. It is, in essence, a requirement that the total latency of all the components in the specified flow path should not exceed the latency value stated in the end-to-end flow specification. For this example, Figure 16 shows the end-to-end flow across the entire system.

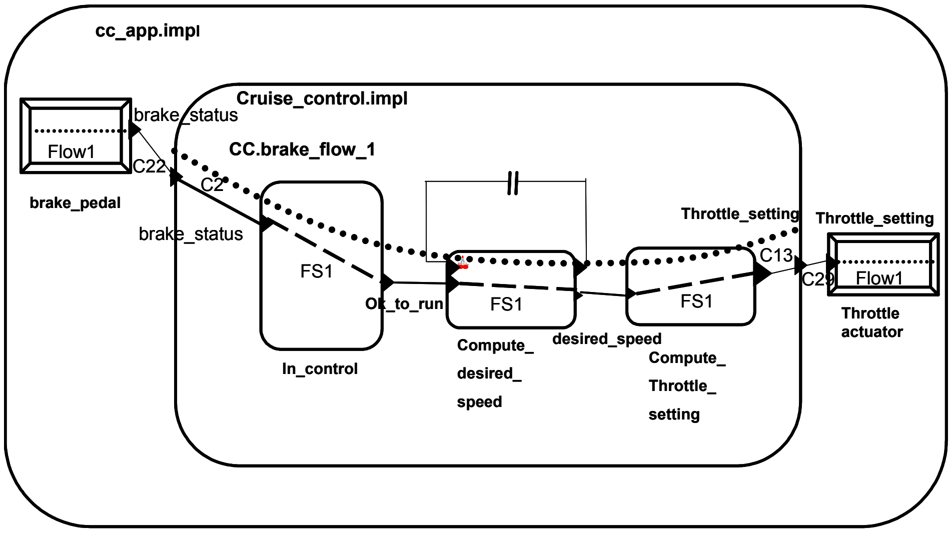


Figure 16: Cruise Control Application Showing End-to-End Flow Path from Brake Pedal to Throttle Actuator

The outermost system oval, CC_app.impl, defines the scope of the entire model, containing all of the devices as well Cruise_control.impl, the application software. The application software subcomponents that make up the processing functions are within Cruise_control.impl. The end-to-end flow specification for CC_appl.impl is formulated by identifying the source device and associated flow, the connection to the application software component (e.g., cruise_control.impl) and its associated flow (CC.brake_flow_1), and the sink device and its associated flow. For this example, the latency requirement for this end-to-end flow path is 200 ms. This end-to-end flow is highlighted in Figure 16 with the dotted line beginning with the brake pedal and ending in the throttle actuator. The code fragment that composes the cc_app.impl is shown below:

```
-- the implementation of the cc system, complete with devices
system implementation cc_app.impl
  -- list of declared components and component implementations
particular to this (cc_app_ implementation
  subcomponents
    CC: system cruise_control.impl;
    BRAKE: device brake_pedal;
    TA: device throttle_actuator;
    CC_ON_OFF: device cruise_control_button;
    ENGINE: device engine;
    RESUME: device resume_button;
    SP_UP: device speed_up_button;
    SP_DN: device speed_dn_button;
    SETBUTTON: device set_button;
    WHEEL_ROT_SENSOR: device wheel_rotation_sensor;
  connections
    -- connect devices to software components
    C21: data port CC_ON_OFF.cc_system_on_off -> CC.cc_system_on_off;
    C22: data port BRAKE.brake_status -> CC.brake_status;
    C23: data port ENGINE.engine_status -> CC.engine_status;
    C24: data port RESUME.resume -> CC.resume;
    C25: data port SP_DN.decrease_speed -> CC.decrease_speed;
    C26: data port SP_UP.increase_speed -> CC.increase_speed;
    C27: data port SETBUTTON.set_speed -> CC.set_speed;
    C28: data port WHEEL_ROT_SENSOR.wheel_pulse -> CC.wheel_pulse;
    C29: data port CC.throttle_setting -> TA.throttle_setting;
  flows
    ETE_F1: end to end flow BRAKE.Flow1 -> C22 -> CC.brake_flow_1
       -> C29 -> TA.Flow1
       {
        Latency => 200 Ms;
        };
end cc_app.impl;
```

In this implementation, devices and software application components that have been declared previously or have implementations are renamed (e.g., the subcomponent `cc_app.impl` has been named `CC`). This renaming is noteworthy because the resulting flows within this implementation must use the name defined within this new implementation. Also, all of the devices are connected to the `cruise_control.impl` via its associated interface ports.

Flow properties are attached to the component declarations. Flow values could also be attached to the implementations. It is suggested that one or the other approach be chosen to avoid possible conflicts of information. Interpreting the presence of multiple flow information for a component declaration and implementation is a matter for the analysis plug-in.

The flow section of `cc_app.impl` contains the end-to-end flow specification, named `ETE_F1` and begins with the source device (as named in this implementation), `BRAKE`, with its associated flow, `Flow1`, the named connection, `C22` to the `cruise_control.impl` (herein named

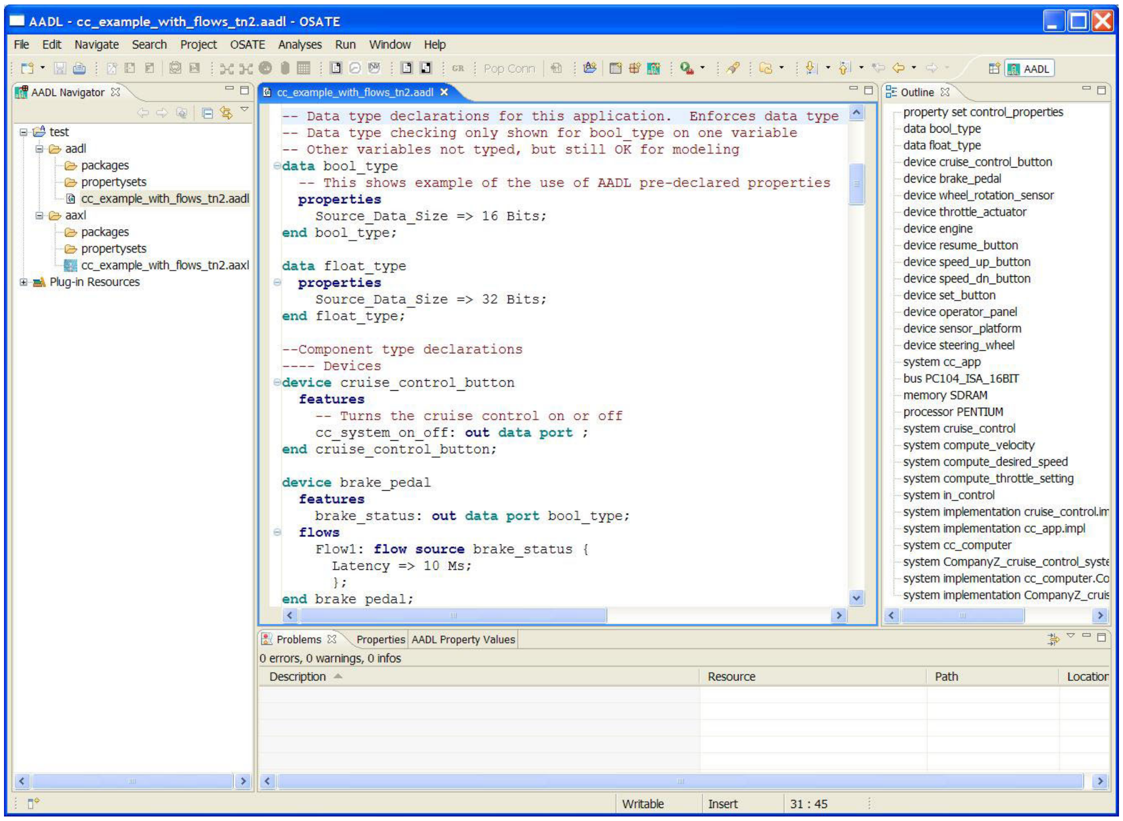
`CC`), the connection `C29` to the throttle actuator device (named `TA`) and its associated flow, `Flow1`.

This discussion completes the detailed modeling of the software and hardware of the embedded cruise control system. The complete model of the cruise control subsystem is in Appendix C (beginning on page 69).

3.6 ANALYSIS

Step 5 of the model development and analysis approach (described on page 13) for this cruise control example is to perform the analysis and evaluate the results. For this example, this step is to determine whether the specified end-to-end latency of the brake-to-throttle actuator signal can be met, given the flow specifications of each component in the flow path.

We entered the components that have been discussed throughout this example into OSATE; the result is shown in Figure 17.


AADL - cc_example_with_flows_tn2.aadl - OSATE
File Edit Navigate Search Project OSATE Analyses Run Window Help
Pop Conn
AADL
AADL Navigator
test
aadl
packages
propertysets
cc_example_with_flows_tn2.aadl
aaxl
packages
propertysets
cc_example_with_flows_tn2.aaxl
Plug-in Resources
cc_example_with_flows_tn2.aadl
Outline

```
-- Data type declarations for this application.  Enforces data type
-- Data type checking only shown for bool_type on one variable
-- Other variables not typed, but still OK for modeling
data bool_type
  -- This shows example of the use of AADL pre-declared properties
  properties
    Source_Data_Size => 16 Bits;
end bool_type;

data float_type
  properties
    Source_Data_Size => 32 Bits;
end float_type;

--Component type declarations
---- Devices
device cruise_control_button
  features
    -- Turns the cruise control on or off
    cc_system_on_off: out data port ;
end cruise_control_button;

device brake_pedal
  features
    brake_status: out data port bool_type;
  flows
    Flow1: flow source brake_status {
      Latency => 10 Ms;
      };
end brake_pedal;
```

```
property set control_properties
data bool_type
data float_type
device cruise_control_button
device brake_pedal
device wheel_rotation_sensor
device throttle_actuator
device engine
device resume_button
device speed_up_button
device speed_dn_button
device set_button
device operator_panel
device sensor_platform
device steering_wheel
system cc_app
bus PC104_ISA_16BIT
memory SDRAM
processor PENTIUM
system cruise_control
system compute_velocity
system compute_desired_speed
system compute_throttle_setting
system in_control
system implementation cruise_control.im
system implementation cc_app.impl
system cc_computer
system CompanyZ_cruise_control_syste
system implementation cc_computer.Co
system implementation CompanyZ_cruis
```

0 errors, 0 warnings, 0 infos

Description	Resource	Path	Locatior

Figure 17: The OSATE Environment Containing the Model of the Cruise Control

The OSATE window contains these four panes:

- The pane on the left-hand side contains the project workspace and associated project files. (This area is known as the AADL Navigator.)
- The middle pane is a multi-tabbed, editable text area that displays the model in the form that the user selects in the AADL Navigator pane (e.g., aadl [source text] or aaxl [compilation of aadl]).
- The right-hand pane shows the outline view of all the components contained in the model.
- The multi-tabbed bottom pane shows problems, properties, and AADL property values.

A detailed description of the mechanics of model composition and analysis using OSATE is beyond the scope of this report, but we will review the functionality of each pane in the context of discussing our example. Once a workspace has been defined, the user defines a new AADL model name, which will appear in the AADL folder in the left-hand pane with the extension .aadl.[2] In our example, the model name is cc_example_with_flows_tn2.aadl. The user enters a textual AADL model in the middle pane. When model entry is completed, the user clicks the disk icon in the upper left-hand side tool bar to run the parser and generate the aaxl instance file. (In our example, this file is named cc_example_with_flows_tn2.aaxl.) The user's action also creates the model object outline in the right-hand pane. Any errors will appear in the Problems tab in the bottom middle of the OSATE window.

With the model entered and no errors (as shown in Figure 17) discovered, the model can be instantiated, as shown in Figure 18, by

1. selecting the implementation System Impl CompanyZ_cruise_control_system.impl name in the middle pane and right-clicking on it
2. clicking to select the *OSATE* command from the resulting list
3. clicking to select the *Instantiate System* command from the list revealed for the *OSATE* command

[2] For information on defining a workspace, see Section 4.1 in the OSATE user manual [SEI AADL 06].

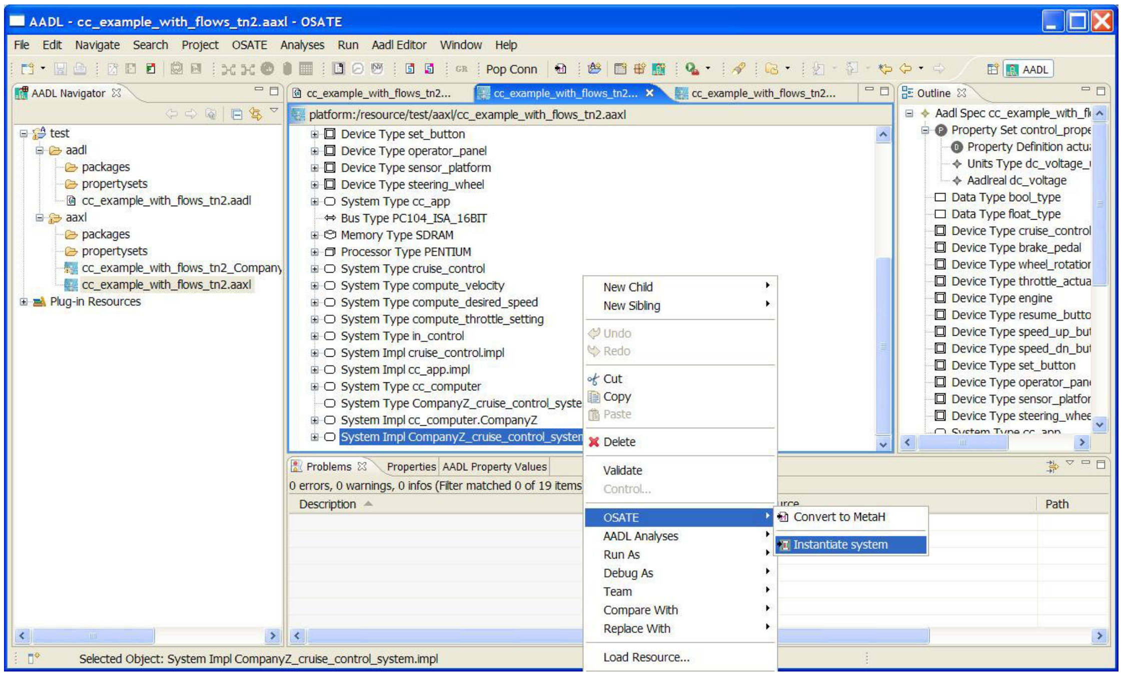
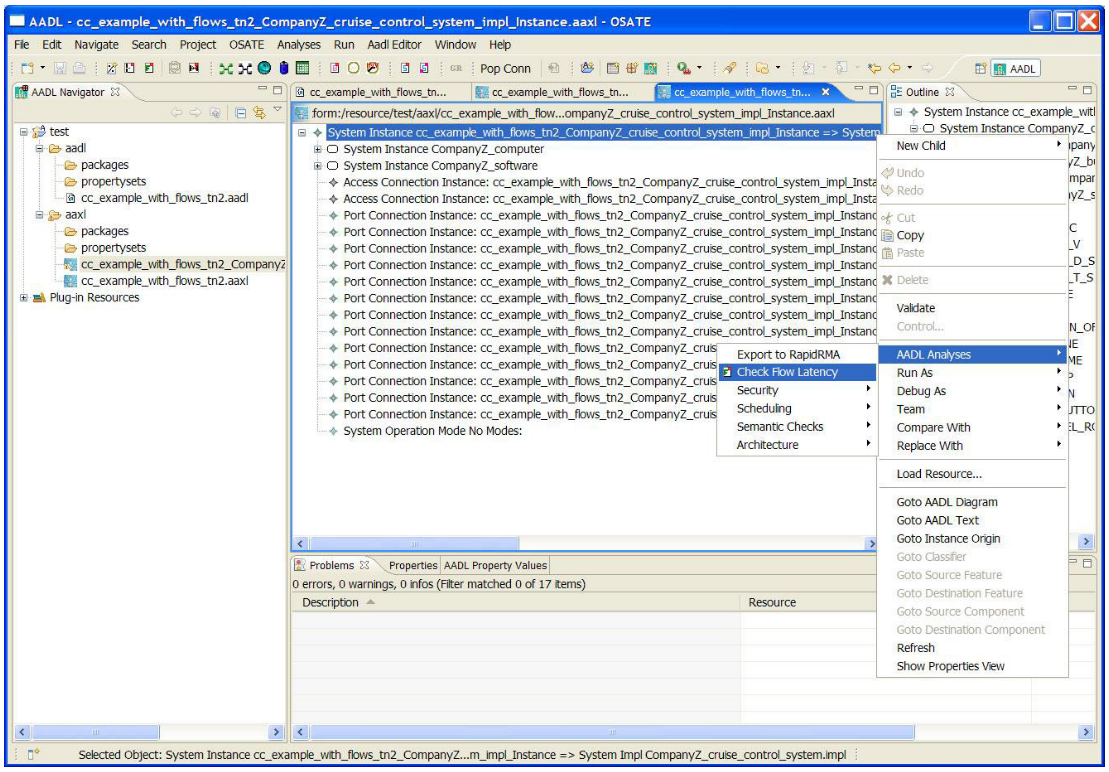

AADL - cc_example_with_flows_tn2.aaxl - OSATE
File Edit Navigate Search Project OSATE Analyses Run Aadl Editor Window Help
AADL Navigator
test
aadl
packages
propertysets
cc_example_with_flows_tn2.aadl
aaxl
packages
propertysets
cc_example_with_flows_tn2_Company
cc_example_with_flows_tn2.aaxl
Plug-in Resources
platform:/resource/test/aaxl/cc_example_with_flows_tn2.aaxl
Device Type set_button
Device Type operator_panel
Device Type sensor_platform
Device Type steering_wheel
System Type cc_app
Bus Type PC104_ISA_16BIT
Memory Type SDRAM
Processor Type PENTIUM
System Type cruise_control
System Type compute_velocity
System Type compute_desired_speed
System Type compute_throttle_setting
System Type in_control
System Impl cruise_control.impl
System Impl cc_app.impl
System Type cc_computer
System Type CompanyZ_cruise_control_syste
System Impl cc_computer.CompanyZ
System Impl CompanyZ_cruise_control_systen
New Child
New Sibling
Undo
Redo
Cut
Copy
Paste
Delete
Validate
Control...
OSATE
AADL Analyses
Run As
Debug As
Team
Compare With
Replace With
Load Resource...
Convert to MetaH
Instantiate system
Outline
Aadl Spec cc_example_with_fl
Property Set control_prope
Property Definition actu
Units Type dc_voltage_
Aadlreal dc_voltage
Data Type bool_type
Data Type float_type
Device Type cruise_control
Device Type brake_pedal
Device Type wheel_rotatior
Device Type throttle_actua
Device Type engine
Device Type resume_butto
Device Type speed_up_but
Device Type speed_dn_but
Device Type set_button
Device Type operator_pan
Device Type sensor_platfor
Device Type steering_whee
Problems
Properties
AADL Property Values
0 errors, 0 warnings, 0 infos (Filter matched 0 of 19 items
Description
Path
Selected Object: System Impl CompanyZ_cruise_control_system.impl

Figure 18: Instantiating the Implementation of the Cruise Control System

Selecting the *Instantiate System* command results in the display of the instance model in the middle pane of the OSATE environment, as Figure 19 illustrates. To run the latency analysis on the instantiated model (as shown in Figure 19)

1. right-click in the middle pane on the cc_example_with_flows_tn2_CompanyZ_cruise_control_system_impl_Instance.aaxl file name
2. click to select the *AADL Analysis* command from the resulting list
3. click to select *Check Flow Latency* command from the list revealed for the *AADL Analysis* command

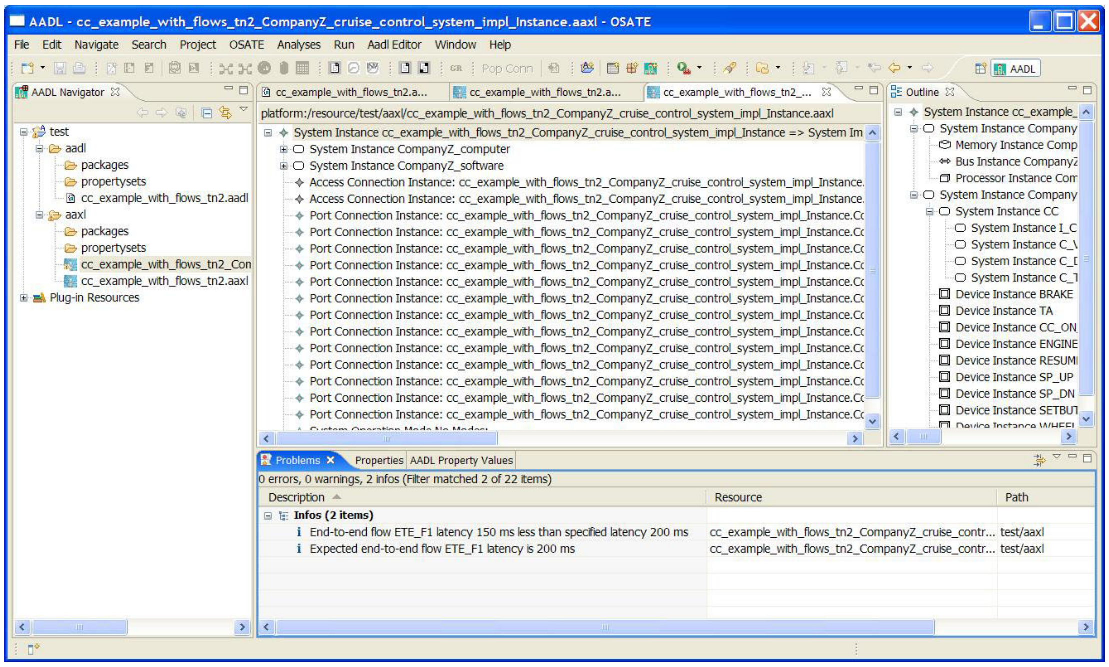

AADL - cc_example_with_flows_tn2_CompanyZ_cruise_control_system_impl_Instance.aaxl - OSATE
File Edit Navigate Search Project OSATE Analyses Run Aadl Editor Window Help
Pop Conn
AADL
AADL Navigator
test
aadl
packages
propertysets
cc_example_with_flows_tn2.aadl
aaxl
packages
propertysets
cc_example_with_flows_tn2_CompanyZ
cc_example_with_flows_tn2.aaxl
Plug-in Resources
form:/resource/test/aaxl/cc_example_with_flow...ompanyZ_cruise_control_system_impl_Instance.aaxl
System Instance cc_example_with_flows_tn2_CompanyZ_cruise_control_system_impl_Instance => System
System Instance CompanyZ_computer
System Instance CompanyZ_software
Access Connection Instance: cc_example_with_flows_tn2_CompanyZ_cruise_control_system_impl_Insta
Port Connection Instance: cc_example_with_flows_tn2_CompanyZ_cruise_control_system_impl_Instanc
System Operation Mode No Modes:
Export to RapidRMA
Check Flow Latency
Security
Scheduling
Semantic Checks
Architecture
New Child
Undo
Redo
Cut
Copy
Paste
Delete
Validate
Control...
AADL Analyses
Run As
Debug As
Team
Compare With
Replace With
Load Resource...
Goto AADL Diagram
Goto AADL Text
Goto Instance Origin
Goto Classifier
Goto Source Feature
Goto Destination Feature
Goto Source Component
Goto Destination Component
Refresh
Show Properties View
Outline
Problems
Properties
AADL Property Values
Description
Resource

0 errors, 0 warnings, 0 infos (Filter matched 0 of 17 items)

Selected Object: System Instance cc_example_with_flows_tn2_CompanyZ...m_impl_Instance => System Impl CompanyZ_cruise_control_system.impl

Figure 19: Running the Check Flow Latency Analysis Plug-in

The result of running the latency analysis plug-in is shown in Figure 20.


AADL - cc_example_with_flows_tn2_CompanyZ_cruise_control_system_impl_Instance.aaxl - OSATE
File Edit Navigate Search Project OSATE Analyses Run Aadl Editor Window Help
AADL Navigator
test
aadl
packages
propertysets
cc_example_with_flows_tn2.aadl
aaxl
packages
propertysets
cc_example_with_flows_tn2_Con
cc_example_with_flows_tn2.aaxl
Plug-in Resources
platform:/resource/test/aaxl/cc_example_with_flows_tn2_CompanyZ_cruise_control_system_impl_Instance.aaxl
System Instance cc_example_with_flows_tn2_CompanyZ_cruise_control_system_impl_Instance => System Im
System Instance CompanyZ_computer
System Instance CompanyZ_software
Access Connection Instance: cc_example_with_flows_tn2_CompanyZ_cruise_control_system_impl_Instance.
Port Connection Instance: cc_example_with_flows_tn2_CompanyZ_cruise_control_system_impl_Instance.C
Outline
System Instance cc_example_
System Instance Company
Memory Instance Comp
Bus Instance CompanyZ
Processor Instance Com
System Instance CC
System Instance I_C
System Instance C_\
System Instance C_[
System Instance C_1
Device Instance BRAKE
Device Instance TA
Device Instance CC_ON
Device Instance ENGINE
Device Instance RESUM
Device Instance SP_UP
Device Instance SP_DN
Device Instance SETBUT
Problems
Properties
AADL Property Values
Description
Resource
Path

0 errors, 0 warnings, 2 infos (Filter matched 2 of 22 items)

Description	Resource	Path
Infos (2 items)		
End-to-end flow ETE_F1 latency 150 ms less than specified latency 200 ms	cc_example_with_flows_tn2_CompanyZ_cruise_contr...	test/aaxl
Expected end-to-end flow ETE_F1 latency is 200 ms	cc_example_with_flows_tn2_CompanyZ_cruise_contr...	test/aaxl

Figure 20: Analysis View of the Cruise Control Instance Model and Latency Analysis Results

Analysis results appear in the pane at the bottom middle of the window, in the Problems tab. The flow implementation for the `ETE_F1` flow path is computed. The flow path of interest (described in Section 3.4.1) is composed of three subcomponents: (1) `in_control` with a latency of `30 ms`, (2) `compute_desired_speed` with a latency of `40 ms`, and (3) `com-pute_throttle_setting` of `50 ms`. Also, the brake pedal device has a latency of `10 ms`, and the throttle actuator device has a latency of `20 ms`. The total of these latencies is 150 ms, which agrees with the analysis results shown in the bottom middle pane in Figure 20. The actual latency of 150 ms is less than the specified latency of 200 ms, indicating that the actual processing time of the brake pedal to throttle actuator path is less than the budgeted amount. A more extensive analysis would entail describing a number of additional flow paths and checking to see if their latencies would be less than or equal to the specified latency. If there were exceptions, the latency requirements could be met by modifying the code to reduce execution time or the system architecture to redefine component connections and perhaps flow paths.

This flow analysis was performed on a high-level architecture whose components are modeled as AADL systems. This model can be refined into a runtime architecture represented by AADL threads. The latency analysis performed on the high-level model determines its latency from the latency values of the system component flow specifications. This analysis can be rerun on the runtime architecture model, in which case the analysis plug-in accounts for worst-case execution time, deadlines, and sampling by periodic threads, and compares the result to the flow specification latencies.

3.6.1 Notes on OSATE

The models developed as part of this report can be found at www.aadl.info. They were run using OSATE V 1.4.6, the latest version at the time this report was prepared. Minor modifications to the model may be necessary for it to run using future versions. The current version of OSATE can be found through http://www.sei.cmu.edu/pcs/model.html, http://www.aadl.info, or http://la.sei.cmu.edu/aadlinfosite/OSATEUpdateSite/.

As referenced in Section 1.1, OSATE is the Eclipse-based open-source tool environment that is the front-end AADL parser and generator. AADL models are developed in OSATE and an intermediate representation (an XML-persistent document) is generated. The intermediate representation communicates the modeled elements to an underlying analysis engine. The intermediate representation is in XML, which allows interoperability among the embedded AADL-XML tool interface, the tool-specific XML representation, and a tool-specific representation. The interface mechanism to accomplish interoperability in Eclipse/OSATE is the plug-in. The plug-in is the standard interface mechanism in the Eclipse environment. Refer to the *OSATE Plug-in Development Guide* [Feiler 06b] for details regarding developing analysis plug-ins and interfacing to external tools. The version of the OSATE current at the time this report was developed contains a small number of plug-ins (for example, computing model statistics and latency analysis).

The AADL modeling language can provide system models to the level of detail required for many types of analysis. The analysis is carried out by the back-end analysis engine via the plug-in interface. Examples of analysis supported by the AADL include, but are not limited to

- scheduling
- throughput
- latency
- resource utilization
- error analysis (e.g., Markov analysis of faulted hardware and software)
- reliability (e.g., failure modes and effects analysis [FMEA] and fault tree [FT] analysis)
- security

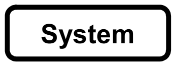
Many commercially developed, open-source, and academically developed analysis tools are available to carry out the desired analysis. As the wider adoption and use of the AADL and OSATE tool environment occur, a body of plug-ins to open source and commercially available analysis tools will evolve.

4 Summary

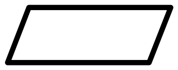
An approach has been presented to model various vehicle systems (e.g., stability control or cruise control) in a hierarchical manner using the SAE AADL. The cruise control system was modeled in detail by modeling each of the computational components as well as sensors and actuators. Included in the model was a flow specification for analyzing end-to-end data-flow latency of a particular signal path through the automotive cruise control. Both the declared and instantiated parts of the model have been developed. The mechanics of connections and flow have been presented, as well as a description of running the latency analysis plug-in and interpreting the results. The source code of the completed model is contained in Appendices B and C. A brief description of using the AADL parser (OSATE) and associated Eclipse tool environment has been discussed.

Appendix A: AADL Graphical Notation

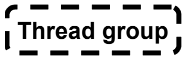
Application Components



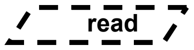
System: hierarchical organization of components



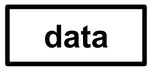
Process: protected address space



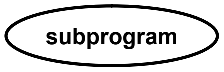
Thread group: organization of threads in processes



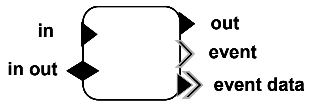
Thread: a schedulable unit of concurrent execution



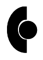
Data: potentially sharable data



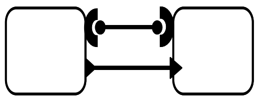
Subprogram: callable unit of sequential code



Port: in, out, in out, event; logical connection points between components

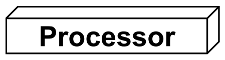


Port group: a group of ports

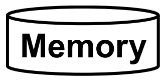


Connection: connects ports in the direction of their flow

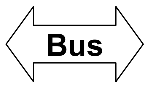
Execution Platform Components



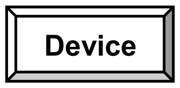
Processor: provides thread scheduling and execution services



Memory: provides storage for data and source code



Bus: provides physical connectivity between execution platform components



Device: interface to external environment

Appendix B: AADL Model of the Vehicle Control Systems

```
-- AADL model of automobile control systems consisting of: traction
-- control, stability control, cruise control, and antilock brake
-- systems.
-- Use with Figure 2: AADL context diagram for a set of vehicle
-- control systems
--
-- THIS IS THE 'PORT GROUP' VERSION

-- data type declarations
data bool_type
-- This shows example of the use of AADL pre-declared properties
   properties
      Source_Data_Size => 16 bits;
end bool_type;

data real_type
   properties
   Source_data_size => 32 bits;
end real_type;

-- Device declarations -------------------------------------------

device engine
   features
      engine_signals: port group engine_socket_1;
-- engine is off or on
end engine;

device wheel_rotation_sensor
   features
      wheel_signal:  port group wheel_sensors_socket_1;
end wheel_rotation_sensor;

device brake_pedal
   features
      brake_signals: port group brake_sensors_socket_1;
end brake_pedal;


device vehicle_state_sensors
   features
          vehicle_state_signals: port group
         vehicle_state_sensors_socket_1;
end vehicle_state_sensors;

device user_console
   features
      user_console_outputs: port group user_console_socket_1;
end user_console;

----- output devices
device throttle_actuator
   features
       tc_throttle_signals: port group tc_throttle_actuator_socket_1;
       cc_throttle_signals: port group cc_throttle_actuator_socket_1;
```

```
end throttle_actuator;

device display
   features
        tc_display_input_signals:  port group
tc_user_display_socket_1;
        cc_display_input_signals:  port group
cc_user_display_socket_1;
        sc_display_input_signals:  port group
sc_user_display_socket_1;
        abs_display_input_signals:  port group
abs_user_display_socket_1;
end display;

device brake_actuators
   features
-- device receives braking signals from three systems
        tc_brake_actuator_signals:  port group
tc_brake_actuator_socket_1;
        sc_brake_actuator_signals:  port group
sc_brake_actuator_socket_1;
        abs_brake_actuator_signals:  port group
abs_brake_actuator_socket_1;

end brake_actuators;


-- port group declarations ----------------------------------

port group  engine_socket_1
   features
      engine_status: out data port;  -- engine is off (0) or on (1)
      engine_temp_1:  out data port;
      throttle_position: out data port;
end engine_socket_1;

port group  engine_plug_1
   inverse of engine_socket_1
end engine_plug_1;


port group  wheel_sensors_socket_1
   features
      wheel_pulse: in data port bool_type;
      wheel_slippage: in data port real_type;
end wheel_sensors_socket_1;

-- assume only one rotation sensor on one wheel....could add one on
-- other wheels for redundancy
port group  wheel_sensors_plug_1
   inverse of wheel_sensors_socket_1
end wheel_sensors_plug_1;

port group  brake_sensors_socket_1
   features
      brake_status: out data port;
end brake_sensors_socket_1;


port group  brake_sensors_plug_1
```

```
   inverse of engine_socket_1
end brake_sensors_plug_1;


port group   vehicle_state_sensors_socket_1
   features
      steering_wheel_angle:  in data port;
      yaw_rate:  in data port;
      lateral_acceleration:  in data port;
end vehicle_state_sensors_socket_1;


port group   vehicle_state_sensors_plug_1
   inverse of vehicle_state_sensors_socket_1
end vehicle_state_sensors_plug_1;


port group   user_console_socket_1
   features
      cc_system_on_off: out data port;
speed_set:  out data port;
      resume:  out data port;
      cancel:  out data port;
      speed_increase: out data port;
      speed_decrease: out data port;
end user_console_socket_1;


port group   user_console_plug_1
   inverse of user_console_socket_1
end user_console_plug_1;

-- TCS output port groups
port group   tc_throttle_actuator_socket_1
   features
      throttle_actuator: out data port;
end tc_throttle_actuator_socket_1;

port group   tc_throttle_actuator_plug_1
   inverse of tc_throttle_actuator_socket_1
end tc_throttle_actuator_plug_1;

port group   tc_user_display_socket_1
   features
      tc_state: out data port;
end tc_user_display_socket_1;

port group   tc_user_display_plug_1
   inverse of tc_user_display_socket_1
end tc_user_display_plug_1;

port group   tc_brake_actuator_socket_1
   features
      tc_brake_output: out data port;
end tc_brake_actuator_socket_1;

port group   tc_brake_actuator_plug_1
   inverse of tc_brake_actuator_socket_1
end tc_brake_actuator_plug_1;
```

```
----------------------------------------
-- cc output port groups
port group   cc_throttle_actuator_socket_1
   features
      cc_throttle_actuator: out data port;
end cc_throttle_actuator_socket_1;

port group   cc_throttle_actuator_plug_1
   inverse of cc_throttle_actuator_socket_1
end cc_throttle_actuator_plug_1;

port group   cc_user_display_socket_1
   features
   cc_state: out data port;
end cc_user_display_socket_1;

port group   cc_user_display_plug_1
   inverse of cc_user_display_socket_1
end cc_user_display_plug_1;

-------------------------------------------------
-- sc output port groups
port group   sc_brake_actuator_socket_1
   features
      sc_brake_actuator: out data port;
end sc_brake_actuator_socket_1;

port group   sc_brake_actuator_plug_1
   inverse of sc_brake_actuator_socket_1
end sc_brake_actuator_plug_1;

port group   sc_user_display_socket_1
   features
      sc_state: out data port;
end sc_user_display_socket_1;

port group   sc_user_display_plug_1
   inverse of sc_user_display_socket_1
end sc_user_display_plug_1;
---------------------------------------------
-- abs output port groups
port group   abs_brake_actuator_socket_1
   features
      abs_brake_actuator: out data port;
end abs_brake_actuator_socket_1;

port group   abs_brake_actuator_plug_1
   inverse of abs_brake_actuator_socket_1
end abs_brake_actuator_plug_1;

port group   abs_user_display_socket_1
   features
      abs_state: out data port;
end abs_user_display_socket_1;

port group   abs_user_display_plug_1
   inverse of abs_user_display_socket_1
end abs_user_display_plug_1;
```

```
system traction_control_system
   features
      tcs_wheel_input: port group wheel_sensors_plug_1;
      tcs_engine_input: port group engine_plug_1;
      tcs_user_input: port group user_console_plug_1;

      tcs_throttle_out: port group tc_throttle_actuator_socket_1;
      tcs_display_out: port group tc_user_display_socket_1;
      tcs_brake_out: port group tc_brake_actuator_plug_1;
end traction_control_system;


system cruise_control_system
   features
      cc_user_input: port group user_console_plug_1;
      cc_wheel_speed: port group wheel_sensors_plug_1;
      cc_engine_input: port group  engine_plug_1;
      cc_brake_status: port group brake_sensors_plug_1;

      cc_throttle_actuator: port group cc_throttle_actuator_plug_1;
      cc_display_out: port group cc_user_display_plug_1;

end cruise_control_system;

system stability_control_system
   features
      sc_user_input: port group user_console_plug_1;
      sc_wheel_speed: port group wheel_sensors_plug_1;
      sc_engine_input: port group  engine_plug_1;
      sc_brake_status: port group brake_sensors_plug_1;

      sc_display_out: port group sc_user_display_plug_1;
      sc_brake_output: port group sc_brake_actuator_plug_1;

end stability_control_system;


system antilock_brake_system
   features
      abs_user_input: port group   user_console_plug_1;
      abs_wheel_speed: port group  wheel_sensors_plug_1;
      abs_engine_input: port group engine_plug_1;

      abs_brake_actuator: port group abs_brake_actuator_plug_1;
      abs_display: port group abs_user_display_plug_1;

end antilock_brake_system;

---------- End of component declarations ----------------------------
-
--- End of example ---------
```

Appendix C: AADL Model of the Cruise Control System

```
-- Cruise Control Example, binding to single CPU, end to end flow
-- model
--
-- Contains the following INPUT devices:
-- CRUISE CONTROL BUTTON, ENGINE, BRAKE_PEDAL, RESUME_BUTTON,
-- SPEED_UP_BUTTON, SPEED_DN_BUTTON, WHEEL_ROTATION_SENSOR
--
-- Contains the following OUTPUT devices:
-- THROTTLE_ACTUATOR
--
-- Contains the following software components:
-- In_control, Compute_desired_speed, Compute_throttle_setting,
-- Compute_velocity
--
-- Analysis views: End-to-end flow of brake pedal -> throttle
-- actuator
-- Compare the declared latency of each component with the latency
-- specified in the implementation.
-- Check to ensure that the actual latency is less than or equal to
-- the specified latency.
-- Software implementation: Single threaded, polled input, sampled
-- output at thread completion time
-- Processor bindings: CompanyZ CPU
--
--
-- Author: J. Hudak
-- Date: January, 2006
-- Revision: August, 13, 2006 - FINAL VERSION for TN
-- User-defined property set for this example
property set control_properties is
  actuator_voltage_range:  range of control_properties::dc_voltage
applies to (data port);
  dc_voltage_units: type units (mV, V => mV * 1000);
  dc_voltage: type aadlreal 0.0 V .. 15.0 V units
control_properties::dc_voltage_units;
end control_properties;

-- Data type declarations for this application.  Enforces data type
-- checking
-- Data type checking only shown for bool_type on one variable
-- Other variables not typed, but still OK for modeling
data bool_type
  -- This shows example of the use of AADL pre-declared properties
  properties
    Source_Data_Size => 16 Bits;
end bool_type;

data float_type
  properties
    Source_Data_Size => 32 Bits;
end float_type;

--Component type declarations
---- Devices
device cruise_control_button
```

```
  features
    -- Turns the cruise control on or off
    cc_system_on_off: out data port ;
end cruise_control_button;

device brake_pedal
  features
    brake_status: out data port bool_type;
  flows
    Flow1: flow source brake_status {
      Latency => 10 Ms;
      };
end brake_pedal;

device wheel_rotation_sensor
  features
    wheel_pulse: out data port ;
end wheel_rotation_sensor;

device throttle_actuator
  features
    throttle_setting: in data port  {
      control_properties::actuator_voltage_range => 0.0 V .. 5.0 V;
      };
  flows
    Flow1: flow sink throttle_setting {
      Latency => 20 Ms;
      };
end throttle_actuator;

-- engine is off (0) or on (1)
device engine
  features
    engine_status: out data port ;
end engine;

-- resume off (0) or on (1)
device resume_button
  features
    resume: out data port ;
end resume_button;

-- increase speed off (0) or increase by 1 mph on (1)
device speed_up_button
  features
    increase_speed: out data port ;
end speed_up_button;

-- decrease speed off (0) or increase by 1 mph on (1)
device speed_dn_button
  features
    decrease_speed: out data port ;
end speed_dn_button;

-- causes the current speed to become the speed setpoint (1)
device set_button
  features
    set_speed: out data port ;
end set_button;
```

```
-- Additional component declarations, not used in connecting to
-- cruise control implementation -------------------------
-- resume off (0) or on (1)
-- TCS operator buttons (TBD)
-- SCS Operator buttons (TBD)
-- ABS Operator buttons - None required
device operator_panel
  features
    -- Cruise Control operator buttons
    cc_system_on_off: out data port bool_type;
    -- turns cc on/off
    set_speed: out data port bool_type;
    -- causes the current speed to become the speed setpoint (1)
    decrease_speed: out data port bool_type;
    -- decrease speed off (0) or increase by 1 mph on (1)
    increase_speed: out data port bool_type;
    -- increase speed off (0) or increase by 1 mph on (1)
    resume: out data port bool_type;
end operator_panel;

-- lateral force of vehicle
device sensor_platform
  features
    -- Sensors used by stability control system
    yaw_sensor_1: out data port float_type;
    -- the instantenous yaw of vehicle
    yaw_rate_sensor_1: out data port float_type;
    -- the yaw rate of the vehicle (x/ms)
    lateral_force_sensor_1: out data port float_type;
end sensor_platform;

-- the steering angle
device steering_wheel
  features
    -- Sensors used by stability control system
    steering_angle: out data port float_type;
end steering_wheel;

------ end of additional component declarations
-- System declarations
-- the cruise control software application is declared (sw + devices,
-- devices will be bound later)
system cc_app
  features
    device_bus: requires bus access PC104_ISA_16BIT;
end cc_app;

-- hardware platform declaration
bus PC104_ISA_16BIT
end PC104_ISA_16BIT;

memory SDRAM
  features
    controller_memory: requires bus access PC104_ISA_16BIT;
end SDRAM;

processor PENTIUM
  features
    controller_cpu: requires bus access PC104_ISA_16BIT;
end PENTIUM;
```

```
-- the cruise control software system - composed of only ports, and
-- the flow we want to analyize
system cruise_control
  features
    cc_system_on_off: in data port ;
    engine_status: in data port ;
    brake_status: in data port bool_type;
    resume: in data port ;
    decrease_speed: in data port ;
    increase_speed: in data port ;
    set_speed: in data port ;
    wheel_pulse: in data port ;
    throttle_setting: out data port ;
 flows
    brake_flow_1: flow path brake_status -> throttle_setting;
end cruise_control;

-- the compute velocity sw component
system compute_velocity
  features
    wheel_pulse: in data port ;
    instantaneous_velocity: out data port ;
  flows
    FS1: flow path wheel_pulse -> instantaneous_velocity;
end compute_velocity;

-- the compute desired speed sw component
system compute_desired_speed
  features
    ok_to_run: in data port ;
    instantaneous_velocity: in data port ;
    current_instantaneous_velocity: out data port ;
    previous_instantaneous_velocity: in data port ;
    desired_speed: out data port ;
  flows
    FS1: flow path ok_to_run -> desired_speed {
      Latency => 40 Ms;
      };
    FS2: flow path instantaneous_velocity -> desired_speed;
end compute_desired_speed;

system compute_throttle_setting
  features
    desired_speed: in data port ;
    throttle_setting: out data port ;
  flows
    FS1: flow path desired_speed -> throttle_setting {
      Latency => 50 Ms;
      };
end compute_throttle_setting;

system in_control
  features
    cc_system_on_off: in data port ;
    brake_status: in data port bool_type;
    resume: in data port ;
    decrease_speed: in data port ;
    increase_speed: in data port ;
    set_speed: in data port ;
    engine_status: in data port ;
    ok_to_run: out data port ;
  flows
    FS1: flow path brake_status -> ok_to_run {
      Latency => 30 Ms;
      };
end in_control;

-- end of the declarations
-- Cruise control implementation
-- Implementation of the cruise control sofware components and
-- connections
system implementation cruise_control.impl
  -- List the declared components that comprise this implementation
  subcomponents
    I_C: system in_control;
    C_V: system compute_velocity;
    C_D_S: system compute_desired_speed;
    C_T_S: system compute_throttle_setting;
  -- make the connections among the components as desired
  -- connections
  -- cruise control port interfaces connected to subcomponent port
  -- interfaces
    C1: data port cc_system_on_off -> I_C.cc_system_on_off;
    C2: data port brake_status -> I_C.brake_status;
    C3: data port engine_status -> I_C.engine_status;
    C4: data port resume -> I_C.resume;
    C5: data port decrease_speed -> I_C.decrease_speed;
    C6: data port increase_speed -> I_C.increase_speed;
    C7: data port set_speed -> I_C.set_speed;
    C8: data port wheel_pulse -> C_V.wheel_pulse;
    C9: data port I_C.ok_to_run -> C_D_S.ok_to_run;
    C10: data port C_V.instantaneous_velocity ->
C_D_S.instantaneous_velocity;
    C11: data port C_D_S.current_instantaneous_velocity ->
C_D_S.previous_instantaneous_velocity;
    C12: data port C_D_S.desired_speed -> C_T_S.desired_speed;
    C13: data port C_T_S.throttle_setting -> throttle_setting;
  -- the flow, begining at the input port, through each of the
  -- necessary subcomponents, to the out port
  flows
    brake_flow_1: flow path brake_status -> C2 -> I_C.FS1
       -> C9 -> C_D_S.FS1
       -> C12 -> C_T_S.FS1
       -> C13 -> throttle_setting;
end cruise_control.impl;

-- the implementation of the cc system, complete with devices
system implementation cc_app.impl
  -- list of declared components and component implementations
  -- particular to this (cc_app_implementation)
  subcomponents
    CC: system cruise_control.impl;
    BRAKE: device brake_pedal;
    TA: device throttle_actuator;
    CC_ON_OFF: device cruise_control_button;
    ENGINE: device engine;
    RESUME: device resume_button;
    SP_UP: device speed_up_button;
    SP_DN: device speed_dn_button;
    SETBUTTON: device set_button;
    WHEEL_ROT_SENSOR: device wheel_rotation_sensor;
  connections
    -- connect devices to software components
    C21: data port CC_ON_OFF.cc_system_on_off -> CC.cc_system_on_off;
    C22: data port BRAKE.brake_status -> CC.brake_status;
    C23: data port ENGINE.engine_status -> CC.engine_status;
    C24: data port RESUME.resume -> CC.resume;
    C25: data port SP_DN.decrease_speed -> CC.decrease_speed;
    C26: data port SP_UP.increase_speed -> CC.increase_speed;
    C27: data port SETBUTTON.set_speed -> CC.set_speed;
    C28: data port WHEEL_ROT_SENSOR.wheel_pulse -> CC.wheel_pulse;
    C29: data port CC.throttle_setting -> TA.throttle_setting;
  flows
    ETE_F1: end to end flow BRAKE.Flow1 -> C22 -> CC.brake_flow_1
       -> C29 -> TA.Flow1
       {
        Latency => 300 Ms;
        };
end cc_app.impl;

system cc_computer
  -- a declaration for cc_computer to be composed of processor,
  -- memory, and bus in its implementation
  -- Needs to provide bus access so the devices in cc_application can
  -- communicate with cpu
  -- Devices need to be attached to a bus.
  features
    device_bus: provides bus access PC104_ISA_16BIT;
end cc_computer;
-- system CompanyZ_computer
-- a declaration for Company Z computer to be composed of processor,
-- memory, and bus in its implementation
-- end CompanyZ_computer;
-- a declaration for Company Z cruise control system to be composed
-- of computer + applications sw in its implementation
system CompanyZ_cruise_control_system
end CompanyZ_cruise_control_system;

-- Cruise control computer implementation specific to CompanyZ
system implementation cc_computer.CompanyZ
  subcomponents
    CompanyZ_memory: memory SDRAM;
    CompanyZ_bus: bus PC104_ISA_16BIT;
    CompanyZ_processor: processor PENTIUM;
  connections
    C1: bus access CompanyZ_bus -> CompanyZ_memory.controller_memory;
    C2: bus access CompanyZ_bus -> CompanyZ_processor.controller_cpu;
end cc_computer.CompanyZ;

-- Composing the complete system
system implementation CompanyZ_cruise_control_system.impl
  subcomponents
    CompanyZ_computer: system cc_computer.CompanyZ;
    CompanyZ_software: system cc_app.impl;
  connections
    C1: bus access CompanyZ_computer.device_bus ->
CompanyZ_software.device_bus;
end CompanyZ_cruise_control_system.impl;
------------------- end of cruise control example ------------------
```

References

URLs are valid as of the publication date of this document.

[Binns 96]
Binns, Pam, Englehart, Matt, Jackson, Mike, & Vestal, Steve. "Domain Specific Software Architectures for Guidance, Navigation and Control." *International Journal of Software Engineering and Knowledge Engineering 6*, 2 (June 1996): 201−227.

[Booch 86]
Booch, Grady. "Object-Oriented Development." *IEEE Transactions on Software Engineering 12*, 2 (February 1986): 211−221.

[Clements 02]
Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little R., Nord, R., & Stafford, J. *Documenting Software Architectures: Views and Beyond* (SEI Series in Software Engineering). Boston, MA: Addison-Wesley, 2002.

[Feiler 00]
Feiler, Peter H., Lewis, Bruce, & Vestal, Steve. *Improving Predictability in Embedded Real-Time Systems* (CMU/SEI-2000-SR-011, ADA387262). Software Engineering Institute, Carnegie Mellon University, 2000.
http://www.sei.cmu.edu/publications/documents/00.reports/00sr011.html

[Feiler 04]
Feiler, Peter H., Lewis, Bruce, Hudak, John, & Gluch, David. *Embedded System Architecture Analysis Using the SAE AADL* (CMU/SEI-2004-TN-005, ADA443481). Software Engineering Institute, Carnegie Mellon University, June 2004.
http://www.sei.cmu.edu/publications/documents/04.reports/04tn005.html

[Feiler 06a]
Feiler, Peter H., Gluch, David P., & Hudak, John J. *The Architecture Analysis & Design Language (AADL): An Introduction* (CMU/SEI-2006-TN-011). Software Engineering Institute, Carnegie Mellon University, 2006.
http://www.sei.cmu.edu/publications/documents/06.reports/06tn011.html

[Feiler 06b]
Feiler, Peter H. & Greenhouse, Aaron. *OSATE Plug-in Development Guide*.
http://www.aadl.info/downloads/Plug-in%20Guide%202006-09-14.pdf (2006).

[Feiler 07]
Feiler, Peter & Rugina, Ana. *Dependability Modeling with AADL* (CMU/SEI-2007-TN-043). Software Engineering Institute, Carnegie Mellon University, 2007.
http:// www.sei.cmu.edu/publications/documents/07.reports/07tn043.html

[Higgins 87]
Higgins, David A. “Specifying Real-Time/Embedded Systems Using Feedback/Control Models,” 127-147. *Proceedings of the Twelfth Structured Methods Conference*. Chicago, IL, August 3−6, 1987. Chicago, IL: Structured Tech. Assoc., 1987.

[McConnell 96]
McConnell, David J., Lewis, Bruce, & Gray, Lisa. “Reengineering a Single Threaded Embedded Missile Application onto a Parallel Processing Platform Using MetaH,” 57−64. *Proceedings of the 5th Workshop on Parallel and Distributed Real Time Systems*. Honolulu, HI, April 15−16, 1996. Los Alamitos, CA: IEEE Computer Society Press, 1996.

[SAE 04]
Society of Automotive Engineers (SAE). *Architecture Analysis & Design Language (AADL)* (Document Number AS5506). Warrendale, PA: Society of Automotive Engineers, November 2004.

[SAE 06]
Society of Automotive Engineers (SAE). *SAE Architecture Analysis and Design Language (AADL) Annex* (Document Number AS5506/1). Warrendale, PA: Society of Automotive Engineers, June 2006.

[SEI AADL 06]
SEI AADL Team. An *Extensible Open Source AADL Tool Environment (OSATE)* (Release 1.3.0, June 2006). Software Engineering Institute, Carnegie Mellon University, 2006. http://la.sei.cmu.edu/aadl/downloads/osate13/AADLToolUserGuide1.3.0%202006-06-02.pdf

[Shaw 95]
Shaw, M. “Beyond Objects: A Software Design Paradigm Based on Process Control.” *ACM SIgsoft Software Engineering Notes 20*, 1 (January 1995): 27−38.

[Smith 88]
Smith, Sharon L. & Gerhart, Susan L. “STATEMATE and Cruise Control: A Case Study.” 49-56. *Proceedings of the Twelfth Annual International Computer Software and Applications Conference (COMPSAC88)*. Chicago, IL, October 5−7, 1988. Washington, DC: IEEE Computer Society Press, 1988.

[Wang 89]
Wang, Jianbai & Tanik, Murat M. “Describing Real Time Systems Using PPA and XYZ/E,” 712-713. *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences, Volume II: Software Track*. Kailua-Kona, HI, January 3−6, 1989. Washington, DC: IEEE Computer Society Press, 1989.

[Ward 84]
Ward, Paul T. & Mellor, Stephen J. “Structured Development for Real Time Systems,” 127−142. *Volume 2, Essential Modeling Techniques*. New York, NY: Yourdon Press, 1984.

[Ward 87]
Ward, Paul. T. & Keskar, Dinesh A. “A Comparison of the Ward/Mellor and Boeing/Hatley Real-Time Methods,” 356−366. *Proceedings of the Twelfth Structured Methods Conference*. Chicago, IL, August 3-6, 1987. Chicago, IL: Structured Tech. Assoc., 1987.

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE July 2007	3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE Developing AADL Models for Control Systems: A Practitioner's Guide		5. FUNDING NUMBERS FA8721-05-C-0003
6. AUTHOR(S) John Hudak, Peter Feiler		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213		8. PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-2007-TR-014
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116		10. SPONSORING/MONITORING AGENCY REPORT NUMBER ESC-TR-2007-014
11. SUPPLEMENTARY NOTES		
12A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS		12B DISTRIBUTION CODE
13. ABSTRACT (MAXIMUM 200 WORDS) This document is a guide to help practitioners using the Architecture Analysis and Design Language (AADL), an international industry standard for the model-based engineering of real-time and embedded systems. The primary goal of this document is to describe an approach for and the mechanics of constructing an architectural model that can be analyzed based on the AADL. The first section of this document presents an overview of AADL concepts and many of the keywords of the language. The second part of the document illustrates a model-building approach using the AADL. It takes the perspective of an engineer who is developing a model for the first time using the AADL. This guide leads the reader through complete AADL model development based on automotive embedded control systems (cruise control, traction control, etc.) by describing the use and syntax of the AADL and interleaving modeling abstraction tradeoffs to achieve models that are abstract but precise. Models are constructed with different analysis perspectives in mind to illustrate the semantics as well as the richness of the AADL.		
14. SUBJECT TERMS Architecture Analysis and Design Language, AADL, architecture model, real-time system, system architecture		15. NUMBER OF PAGES 84
16. PRICE CODE		

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL