The Architecture Analysis & Design Language (AADL): An Introduction

3 AUTHORS, INCLUDING:

Peter H. Feiler
East China Normal University

69 PUBLICATIONS   **1,229** CITATIONS

John Hudak
Carnegie Mellon University

14 PUBLICATIONS   **261** CITATIONS

The Architecture Analysis & Design Language (AADL): An Introduction

Peter H. Feiler
David P. Gluch
John J. Hudak

February 2006

Performance-Critical Systems

Contents

List of Figures

List of Tables

Abstract

In November 2004, the Society of Automotive Engineers (SAE) released the aerospace standard AS5506, named the Architecture Analysis & Design Language (AADL). The AADL is a modeling language that supports early and repeated analyses of a system's architecture with respect to performance-critical properties through an extendable notation, a tool framework, and precisely defined semantics.

The language employs formal modeling concepts for the description and analysis of application system architectures in terms of distinct components and their interactions. It includes abstractions of software, computational hardware, and system components for (*a*) specifying and analyzing real-time embedded and high dependability systems, complex systems of systems, and specialized performance capability systems and (*b*) mapping of software onto computational hardware elements.

The AADL is especially effective for model-based analysis and specification of complex real-time embedded systems. This technical note is an introduction to the concepts, language structure, and application of the AADL.

1 Introduction

This document, Part 1 of a use guide for the Architecture Analysis & Design Language (AADL), provides an introduction to the language and AADL specifications.[1] The AADL is defined in the Society of Automotive Engineers (SAE) standard AS5506.[2]

1.1 Document Summary

Readers who are unfamiliar with the AADL will be able to gain a fuller understanding of the purpose, capabilities, notation, and elements of this modeling language. Table 1-1 summarizes the content in this document.

Table 1-1: Summary of Content in this Document

Section Number	Content Summary
2	Section 2 summarizes the AADL language and introduces the AADL as a framework for the design and analysis of the architectures of component-based systems.
3	Section 3 provides a foundation for more detailed and problem-oriented material in other sections of the document. This section also presents a conceptual overview of the AADL abstractions; subsequent sections supply details on the syntax and semantics of various language constructs.
4	Section 4 focuses on an AADL textual (natural language) specification as a human-readable set of representations that consists of a collection of textual declarations that comply with the AADL standard [SAE 06a]. The graphical representations associated with the textual declarations are also included throughout this document to highlight the relationship between the representations.
5	Section 5 presents the software component abstractions (process, thread, thread group data, and subprogram) and provides example declarations for these components.
6	Section 6 provides the execution platform component abstractions (processor, memory, bus, and device) and provides example declarations for these components.
7	Section 7 discusses the system abstraction and presents examples of the specification of composite systems and their instances.

---

[1] The use guide for the AADL will be published initially as a series of technical notes.

[2] For more information on the development, ongoing applications, and future plans of the AADL, go to http://www.aadl.info. To purchase a copy of the standard, go to http://www.sae.org/servlets /productDetail?PROD_TYP=STD&PROD_CD=AS5506.

Table 1: Summary of Content in this Document (cont.)

Section Number	Content Summary
8	Section 8 describes the abstractions that support the specification of component interactions. Examples of the specification of component interfaces and their interconnections are presented.
9	Section 9 presents the specification of alternative operational states of a system. Modes mode transitions, and examples of their specification are described.
10	Section 10 describes the use of the AADL flows concept and presents examples of the specification of abstract flows throughout a system.
11	Section 11 discusses property constructs and presents examples of property type and name definitions, property set declarations, and property associations.
12	Section 12 describes the constructs for organizing an AADL specification. It includes examples of AADL architectural pattern sets.

The Appendix (pages 117–125) provides tabular summaries of the features, components, and built-in properties of the language.

1.2   Reader's Guide to Technical Interests

Readers familiar with the AADL standard document will be able to take advantage of the detailed descriptions and examples (in textual and graphical forms) shown in the technical interest areas that are correlated with sections in this document in Table 1-2.

Table 1-2: Technical Interests and Relevant Sections in this Document

Section Numbers	Technical Considerations
5.4, 5.5, 8.3.1, 8.3.2, 8.4, and 8.5	Modeling Application Software—These sections address data and subprogram components and their interactions (e.g., calls and component access.
5.1, 5.2, 5.3, 8.1, 8.2, 8.3.1, 8.3.2, and 8.4.2	Execution Tasking and Concurrency—These sections present relevant aspects of runtime interaction, coordination, and timing associated with multiple execution paths.
6, 7, and 8.3.3	System Instances and Binding Software to Hardware Components—These sections discuss issues and capabilities in defining a complete instance of a system architecture.
11	Properties of Model Elements—This section discusses assigning values to properties and defining new properties within an AADL model.

Table 1-2:   Technical Interests and Relevant Sections in this Document (cont.)

Section Numbers	Technical Considerations
9 and 11.2	Partitioning Runtime Configurations—These sections present the structuring of alternative architectural configurations for a system.
10, 11.3, 11.4, and 11.5	Analysis Abstractions—These sections discuss capabilities that facilitate analysis of a system architecture.

1.3   Conventions Used in this Document

The textual and graphical illustrations used in this technical note reflect the styles used in the AADL standard document [SAE 06a], except where noted. In addition, for consistency and clarification in this document, we have represented AADL core language concepts and key specification elements the same way (i.e., using the same type style and format) in textual examples and explanatory text (in sections 4 through 12). Also, we have used the AADL icon (⚏) to indicate a different semantics than that represented by a similar graphical symbol in the Unified Modeling Language (UML).

2   SAE AADL Overview

The SAE AADL standard provides formal modeling concepts for the description and analysis of application systems architecture in terms of distinct components and their interactions. The AADL includes software, hardware, and system component abstractions to

- specify and analyze real-time embedded systems, complex systems of systems, and specialized performance capability systems
- map software onto computational hardware elements

The AADL is especially effective for model-based analysis and specification of complex real-time embedded systems.

2.1   Abstraction of Components

Within the AADL, a component is characterized by its identity (a unique name and runtime essence), possible interfaces with other components, distinguishing properties (critical characteristics of a component within its architectural context), and subcomponents and their interactions.

In addition to interfaces and internal structural elements, other abstractions can be defined for a component and system architecture. For example, abstract flows of information or control can be identified, associated with specific components and interconnections, and analyzed. These additional elements can be included through core AADL language capabilities (e.g. defining new component properties) or the specification of a supplemental annex language.[3]

The component abstractions of the AADL are separated into three categories:

1. application software

   a. thread: active component that can execute concurrently and be organized into thread groups

   b. thread group: component abstraction for logically organizing thread, data, and thread group components within a process

   c. process: protected address space whose boundaries are enforced at runtime

   d. data: data types and static data in source text

   e. subprogram: concepts such as call-return and calls-on methods (modeled using a subprogram component that represents a callable piece of source code)

---

[3]   *Annex libraries* enable a designer to extend the language and customize an AADL specification to meet project- or domain-specific requirements. An *annex document* is an approved extension to the core AADL standard. [SAE 06a].

2. execution platform (hardware)

    a. processor: schedules and executes threads

    b. memory: stores code and data

    c. device: represents sensors, actuators, or other components that interface with the external environment

    d. bus: interconnects processors, memory, and devices

3. composite

    a. system: design elements that enable the integration of other components into distinct units within the architecture

System components are composites that can consist of other systems as well as of software or hardware components.

The AADL standard includes runtime semantics for mechanisms of exchange and control of data, including

- message passing

- event passing

- synchronized access to shared components

- thread scheduling protocols

- timing requirements

- remote procedure calls

In addition, dynamic reconfiguration of runtime architectures can be specified using operational modes and mode transitions.

2.2 Architectural Analysis

The AADL can be used to model and analyze systems already in use and design and integrate new systems. The AADL can be used in the analysis of partially defined architectural patterns (with limited architectural detail) as well as in full-scale analysis of a complete system model extracted from the source code (with completely quantified system property values).

AADL supports the early prediction and analysis of critical system qualities—such as performance, schedulability, and reliability. For example, in specifying and analyzing schedulability, AADL-supported thread components include the predeclared[4] execution property options of periodic, aperiodic (event-driven), background (dispatched once and executed to completion), and sporadic (paced by an upper rate bound) events. These thread characteristics are defined as part of the thread declaration and can be readily analyzed.

---

4   There is a standard predeclared property set named `AADL_Properties` that is part of every AADL specification [SAE 06a].

Within the core language, property sets can be declared that include new properties for components and other modeling elements (e.g. ports and connections). By utilizing the extension capabilities of the language, too, additional models and properties can be included. For example, a reliability annex can be used that defines reliability models and properties of components facilitating a Markov or fault tree analysis of the architecture [SAE 06b]. This analysis would assess an architecture's compliance with specific reliability requirements.

Collectively, these AADL properties and extensions can be used to incorporate new and focused analyses at the architectural design level. These analyses facilitate tradeoff assessments among alternative design options early in a development or upgrade process.

AADL components interact exclusively through defined interfaces. A component interface consists of directional flow through

- data ports for unqueued state data

- event data ports for queued message data

- event ports for asynchronous events

- synchronous subprogram calls

- explicit access to data components

Interactions among components are specified explicitly. For example, data communication among components is specified through connection declarations. These can be midframe (immediate) communication or phase-delayed (delayed) communication. The semantics of these connections assures deterministic transfer of data streams. Deterministic transfer means that a thread always receives data with the same time delay; if the receiving thread is over- or under-sampling the data stream, it always does so at a constant rate.

Application components have properties that specify timing requirements such as period, worst-case execution time, deadlines, space requirements, arrival rates, and characteristics of data and event streams. In addition, properties identify the following:

- source code and data that implement the application component being modeled in the AADL

- constraints for binding threads to processors, source code, and data onto memory

The constraints can limit binding to specific processor or memory types (e.g., to a processor with DSP support) as well as prevent colocation of application components to support fault tolerance [Feiler 04].

3 AADL Language Abstractions

The core language concepts and key specification elements of AADL are summarized in Figure 3-1. In AADL, components are defined through type and **implementation** declarations. A *Component Type* declaration defines a component's interface elements and externally observable attributes (i.e., features that are interaction points with other components, flow specifications, and internal property values). A *Component Implementation* declaration defines a component's internal structure in terms of **subcomponents**, subcomponent **connections**, **subprogram** call sequences, **modes**, **flow** implementations, and **properties**. *Components* are grouped into application software, execution platform, and composite categories. *Packages* enable the organization of AADL elements into named groups. *Property Sets* and *Annex Libraries* enable a designer to extend the language and customize an AADL specification to meet project- or domain-specific requirements.[5]
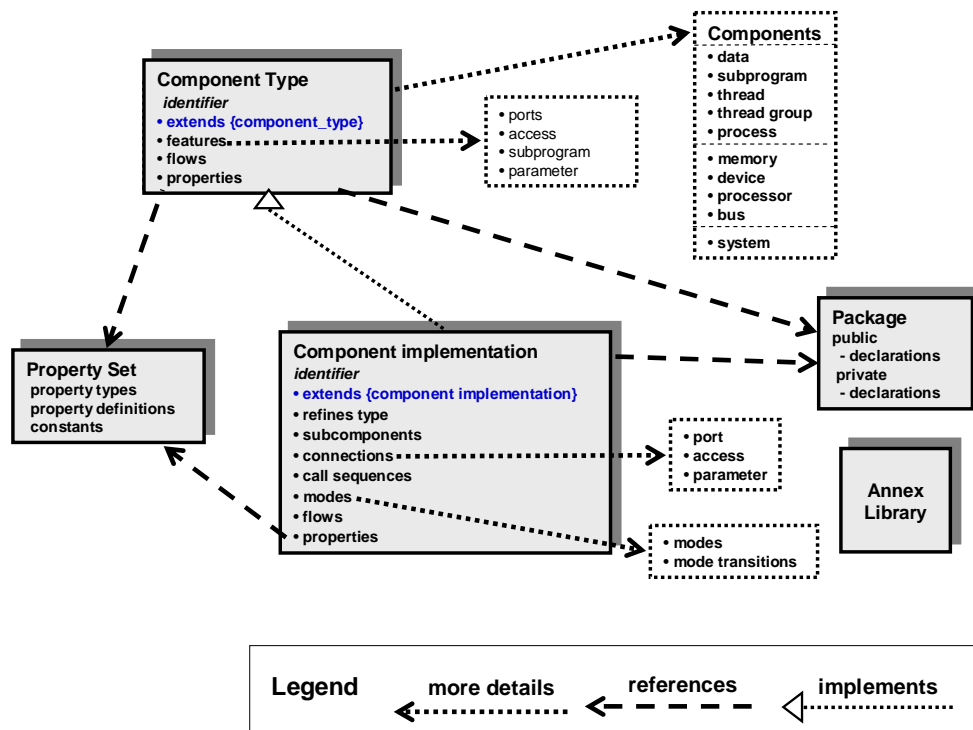
Figure 3-1:   Summary of AADL Elements

---

[5]   *Annex libraries* enable a designer to extend the language and customize an AADL specification to meet project- or domain-specific requirements. An *annex document* is an approved extension to the core AADL standard.

3.1 Components

Components form the central modeling vocabulary for the AADL. Components are assigned a unique identity (name) and are declared as a type and **implementation** within a particular component category. A component category defines the runtime essence of a component. There are three distinct sets of component categories:

1. application software
    a. thread: a schedulable unit of concurrent execution
    b. thread group: a compositional unit for organizing threads
    c. process: a protected address space
    d. data: data types and static data in source text
    e. subprogram: callable sequentially executable code
2. execution platform
    a. processor: components that execute threads
    b. memory: components that store data and code
    c. device: components that interface with and represent the external environment
    d. bus: components that provide access among execution platform components
3. composite
    a. system: a composite of software, execution platform, or system components

Each of the component categories is discussed in separate sections of this document. The syntax and semantics of declarations in an AADL specification are discussed in Section 4.1.

3.2 Component Types

An AADL component type declaration establishes a component's externally visible characteristics. For example, a declaration specifies the interfaces of a **thread** component. A component type declaration consists of a defining clause and descriptive subclauses; Figure 3-2 shows a type declaration of a **thread**. **Features** are the interfaces of the component. **Flows** specify distinct abstract channels of information transfer. **Properties** define intrinsic characteristics of a component. There are predefined **properties** for each component category (e.g., the execution time for a **thread**).

```
thread <name>
   extends
   features
   flows
   properties
```
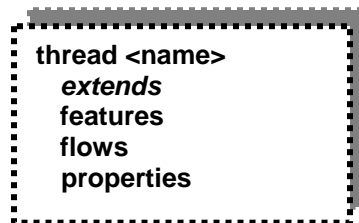
Figure 3-2:   Subclauses of a Type Declaration

The **_extends_** subclause enables one component type declaration to build upon another. A component declared as an extension inherits the characteristics of the original component (i.e., it is a subclass of the original). Within a component declared as an extension of another type, interfaces, flows, and properties can be added; partially declared elements of the antecedent component type can be detailed; and properties can be modified (refined). These qualities permit the modeling of variations in the interfaces of a family of related components.

3.3 Component Implementations

A component **implementation** specifies an internal structure in terms of **subcomponents**, interactions (**calls** and **connections**) among the **features** of those **subcomponents**, **flows** across a sequence of **subcomponents**, **modes** that represent operational states, and **properties**.

The subclauses of an **implementation** declaration are summarized in Figure 3-3. The **subcomponents**, **connections**, and **calls** declarations specify the composition of a component as a collection of components (**subcomponents**) and their interactions. **Flows** represent implementations of flow specifications in the component type or end-to-end flows to be analyzed (i.e., flows that start in one subcomponent, go through zero or more subcomponents, and end in another subcomponent). **Modes** represent alternative operational modes that may manifest themselves as alternate configurations of **subcomponents**, **calls** sequences, **connections**, **flow** sequences, and **properties**. **Properties** define intrinsic characteristics of a component. There are predefined **properties** for each component **implementation**.

```
thread implementation <typeidentifier>.<implementationidentifier>
    extends
    refines type
    subcomponents
    calls
    connections
    flows
    modes
    properties
```
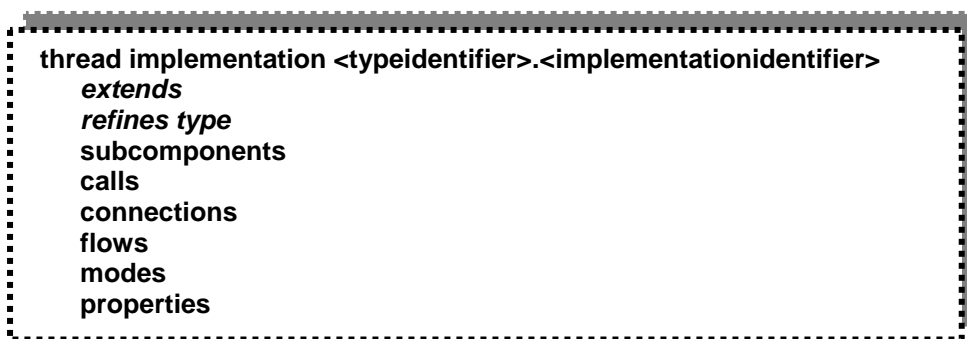
Figure 3-3:   Subclauses of an Implementation Declaration

Multiple implementations of a component type can be declared, allowing multiple variants with the same external interfaces to be modeled because each **implementation** provides a realization of a component that satisfies the same interface specified by the component type. In addition, a component **implementation** may extend and refine other previously declared component implementations. Extended implementations (declared with the **_extends_** subclause) inherit the characteristics of the original component **implementation** and all of its predecessors. Refinement allows partially specified

component implementations (templates) to be completed, while extension allows a component **implementation** to be expressed as variation of a common component description through additions. In addition, an ***extends* implementation** declaration can add **property** values to the **features** of its corresponding type. These additions can be made through the ***refines type*** subclause.

Component decomposition is defined through **subcomponents** declarations within component **implementation** declarations. A subcomponent represents the decomposition element and the **classifier** (named **implementation**) represents a choice in a family. A component instance is created by instantiating a component **implementation** and each of its subcomponents recursively.

3.4 Packages, Property Sets, and Annexes

AADL packages permit collections of component declarations to be organized into separate units with their own namespaces. Elements with common characteristics (e.g., all components associated with network communications) can be grouped together in a **package** and referenced using the **package** name. Packages can support the independent development of AADL models for different subsystems of a large-scale **system** by providing a distinct namespace for each group of subsystem elements.

A property set is a named grouping of **property** declarations that define new **properties** and **property** types that can be included in a specification. For example, a security **property set** can include definitions for security levels required in a database system. These **properties** are referenced using the **property set** name and can be associated with components and other modeling elements (e.g., ports or connections) within a system specification. Their declaration and use become part of the specification.

An **annex** enables a user to extend the AADL language, allowing the incorporation of specialized notations within a standard AADL model. For example, a formal language that enables an analysis of a critical aspect of a system (e.g., reliability analysis, security, or behavior) can be included within an AADL specification.[6]

Each of these elements is described in more detail in other sections of this document.

---

[6] *Annex libraries* enable a designer to extend the language and customize an AADL specification to meet project- or domain-specific requirements. An *annex document* is an approved extension to the core AADL standard.

4   AADL System Models and Specifications

An AADL system model describes the architecture and runtime environment of an application system in terms of its constituent software and execution platform (hardware) components and their interactions. An AADL model is captured in a specification consisting of syntactically and semantically correct AADL declarations. A complete AADL system model includes all of the declarations required to instantiate a runtime instance of an application system that the specification represents (e.g., an aircraft's flight control system).

From a user perspective, an AADL specification and its constituent declarations can be expressed textually, graphically, in a combination of those representations, or as Extensible Markup Language (XML). The AADL textual and graphical notations are defined by the SAE AADL standard and its extensions [SAE 06a]. The XML form is defined in *Extensible Markup Language (XML) 1.0 (Third Edition)* [W3C 04]. Figure 4-1 summarizes the alternative representations of an AADL specification, showing sample textual, graphical, and XML representations.
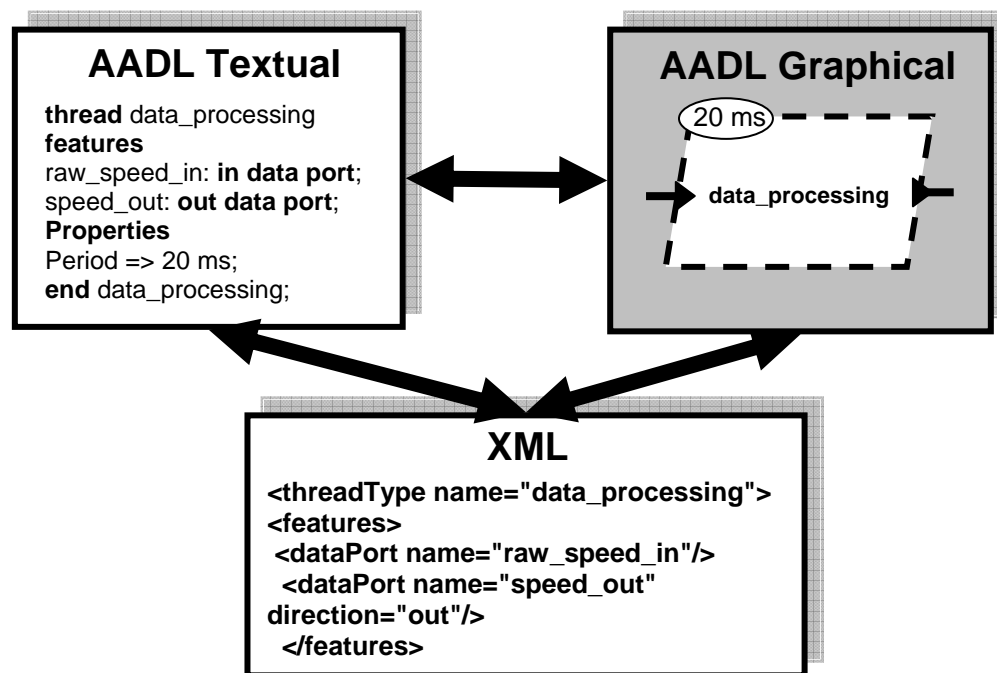
Figure 4-1:   AADL Representations

4.1 AADL Textual Specifications

This section focuses on an AADL textual (natural language) specification as a human-readable collection of textual declarations that comply with the AADL standard [SAE 06a]. Graphical notations associated with the textual specifications are included in this document to highlight the relationship between representations and to help the reader visualize the architecture. Detailed descriptions of the graphical representations for AADL constructs and declarations are provided in the graphical standard.[7] The principal AADL declarations are summarized in Table 4-1.

Table 4-1: Principal AADL Declarations

Declaration	Description
Component Type: system, process, thread, thread group data, subprogram, processor, device, memory, and bus	The component type declaration establishes the identity (component category and name) and defines the features, flows, and properties of a component type. A component type declaration may also declare the type as an extension of another type (extends).
Component Implementation: system, process, thread, thread group data, subprogram, processor, device, memory, and bus	The component implementation declaration establishes the identity (component category, type, and name) and defines the refinements (refines type subclause), subcomponents, calls, connections, flows, modes, and properties of a component implementation. The identity must include a declared component type consistent with the component category. The component implementation declaration may also declare the implementation as an extension of another implementation (extends subclause).
Port Group Type	Port group type declarations establish the identity (name) and define the features and properties of a grouping of ports and/or port groups. Within the features declaration, a port group may be defined as the inverse of another port group. A port group type declaration may also declare the port group as an extension of another port group type (extends).
Package	The package declaration establishes the identity (name) of a collection of AADL declarations, groups those declarations into private and public sections, and declares properties associated with a package. Packages are used to logically organize AADL declarations. AADL component type, implementation, and port group declarations placed in AADL packages can be referenced by declarations in other packages.

---

[7]   The complete set of graphical symbols for AADL components is presented in "Graphical AADL Notation," a draft document at the time of the publishing of this technical note.

Table 4-1: AADL Declarations (cont.)

Property Set	Property set declarations introduce additional properties, property types, and property constants that are not included as predeclared AADL properties.[8] Each property set has a unique global name and provides a unique namespace for the items declared in it. In other words, properties and property types declared in a property set are referenced by property set name and item name.
Annex Library	Annex library declarations establish the identity (name) and define the contents of a set of reusable declarations that are not part of the standard AADL language. Annex declarations are used to extend AADL's core modeling and analysis capabilities.

4.2 Graphical Representations

The AADL's graphical notation facilitates a clear visual presentation of a system's structural hierarchy and communication topology and provides a foundation for distinct architecture perspectives. Graphical notation elements for AADL components are shown in Figure 4-2. The letter-shaped AADL icon (⚛) is used to indicate a different semantics than that represented by a similar graphical symbol in the Unified Modeling Language (UML). This symbol is not required in notation; it can be used where a distinction from a UML symbol is necessary. Additional symbols, such as circles, are used to represent component properties (e.g., the period of a **thread**).
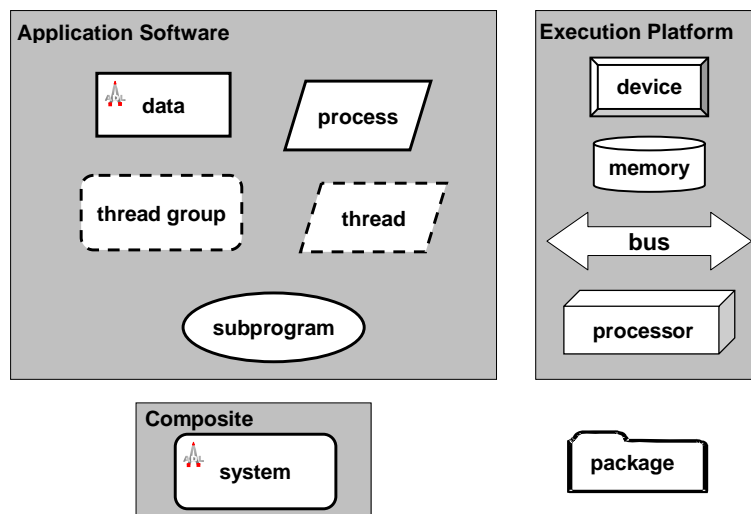
Figure 4-2: AADL Graphical Notation

---

[8] There is a standard predeclared property set named `AADL_Properties` that is a part of every AADL specification [SAE 06a].

4.3   Example Specification

Table 4-2 contains an excerpt from an AADL textual specification and includes sample graphical representations of portions of the specification.[9] The excerpt shows simplified component type, component **implementation**, and **subcomponents** declarations (i.e., only some of the **features**, **flows**, or **properties** are declared) and illustrates the pattern other examples in this document will follow.

In the example shown in Table 4-2, related type and **implementation** declarations are grouped together. Individual declarations can be arranged in any order within an AADL specification. For example, a component type declaration that includes a specific **port group** as one of its interfaces (**features**) can precede that port group's type declaration. An alternative organization might involve grouping together all type declarations. In addition, all or some of the declarations shown in Table 4-2 can be partitioned into groups using **packages**. The options provided by **packages** and their implications are discussed in Section 12 (Organizing a Specification).

The excerpt in Table 4-2 contains one **process** and two **thread** component type declarations. The **process** type definition has the component type identifier (name) `control_processing`. Two data ports, **in data port** and **out data port**, are declared for this **process** type. The `sensor_data` and `command_data` data types are declared in individual **data** type declarations.

The **thread** type definition identifiers are `control_in` and `control_out`. An **implementation** declaration of the **process** type `control_processing` is shown. The component **implementation** identifier is `speed_control`. An **implementation** is referenced by using both the component type identifier and the component **implementation** identifier, separated by a period (.). A reference to a **thread implementation** `input_processing_01` of the **thread** type `control_in` is shown in the declaration of the subcomponent `control_input`. Thus, `control_input` is an instance of the component **implementation** `control_in.input_processing_01`.

Graphical representations of the **process** type declaration `control_processing` and the **process implementation** declaration are shown in the latter portions of Table 4-2. The **process implementation** symbol in the example is bounded with a bold line. Bold-lining of an **implementation** symbol is optional. It can be useful in distinguishing component type and component **implementation** representations visually. A solid black triangular symbol represents a **data port.** Port and other **features** symbols are discussed in Section 8 (Component Interactions).

---

9   In the example specifications shown here and in Sections 5–12, we typically follow the pattern of displaying the textual representation followed by the graphical representation in portions of the same table, as shown in Table 4-2. Where needed to provide clarification, we have placed the textual and graphical representations in separate tables and figures.

Table 4-2: A Simplified Example of an AADL Specification[10]

```
-- A process type definition with the component type
-- identifier (name) "control_processing" is shown below.

process control_processing
features
input: in data port sensor_data;
output: out data port command_data;
end control_processing;

-- Below is an implementation of process type "control_processing"
-- The component implementation identifier(name)is "speed_control"
-- The implementation is referenced by using both the component type
-- identifier and the component implementation identifier, separated
-- by a period(.)in the form: control_processing.speed_control.
-- A reference to a thread implementation "input_processing_01"
-- of the thread type "control_in" is shown below in the
-- declaration of the subcomponent "control_input"

process implementation control_processing.speed_control
subcomponents
control_input: thread control_in.input_processing_01;
control_output: thread control_out.output_processing_01;
end control_processing.speed_control;

-- The declaration of the thread type "control_in" is shown below.
thread control_in
end control_in;

-- The declaration of the thread implementation
-- "control_in.input_processing_01" is shown below.
thread implementation control_in.input_processing_01
end control_in.input_processing_01;

-- The declaration of the thread type "control_out" is shown below.
thread control_out
end control_out;

-- The declaration of the thread implementation
-- "control_out.output_processing_01" is shown below.
thread implementation control_out.output_processing_01
end control_out.output_processing_01;

-- The declaration of the data type "sensor_data" is shown below.
data sensor_data
end sensor_data;

-- The declaration of the data type "command_data" is shown below.
data command_data
end command_data;
```

---

[10]   Comment lines in an AADL specification are prefaced by two dashes (--).

Table 4-2: A Simplified Example of an AADL Specification (cont.)

process type control_processing

process implementation control_processing.speed_control
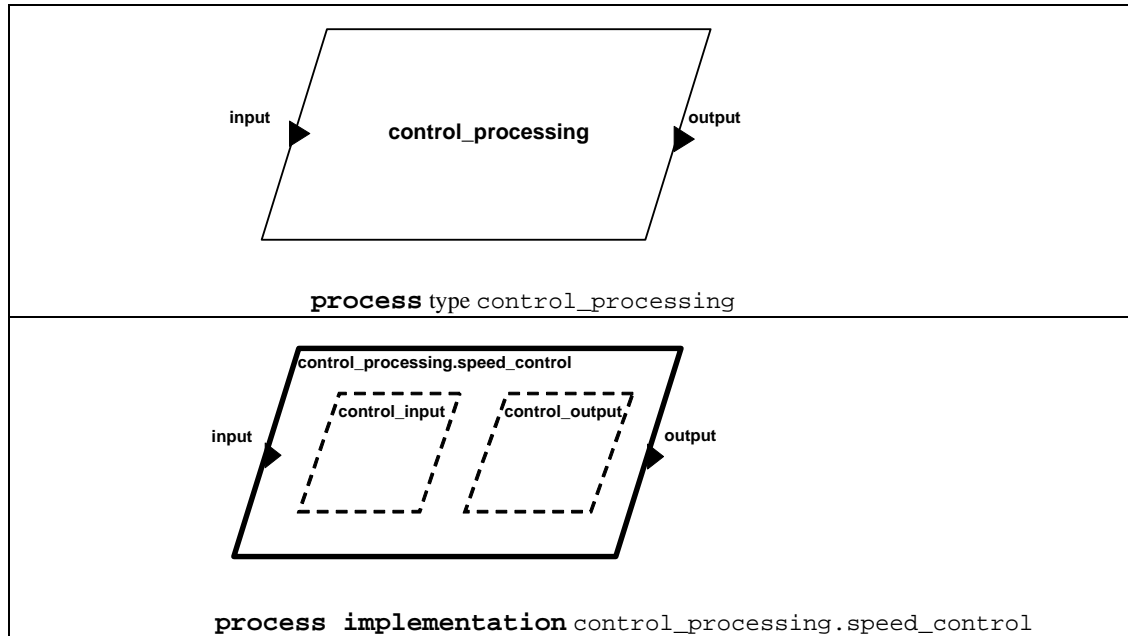
4.4   Type Declarations

The structures for a component type declaration (area labeled ①) and a type declaration that extends another type (area labeled ②) are shown in Table 4-3, along with sample component type declarations (area labeled ③). The sample type declarations are for a **process** type simple_speed_control and a **thread** type data_management. The first line of each declaration begins with the appropriate component category reserved word in boldface. In these examples, **process** and **thread** are reserved words.

Table 4-3: Sample Component Type Declarations

```
component_category type_identifier                              ①
  features
  flows
  properties
end type_identifier ;
```

```
component_category type_identifier                              ②
  extends unique_component_type_identifier
  features
  flows
  properties
end type_identifier ;
```

Table 4–3:   Sample Component Type Declarations (cont.)

```
process simple_speed_control                                    ③
features
raw_speed: in data port speed_type;
toggle_mode: in event port;
throttle_cmd: out data port throttle_data;
flows none;
end simple_speed_control;

thread data_management extends system_management
features
in_data: refined to in data port speed_type;
out_data: out data port throttle_data;
end data_management;

data speed_type
end speed_type;

data throttle_data
end throttle_data;

thread system_management
features
in_data: in data port;
end system_management;
```

The component type **classifier** (name) of the type follows the component category reserved word. A component type declaration may contain up to four subclauses that are identified with these reserved words:

- **features**: specifies the interaction points with other components, including the inputs and accesses required by the component and all the outputs and items the component provides

- **flows**: defines specifications of logical flows through the component from incoming interaction points to outgoing interaction points (These flows can be used to specify end-to-end flows without having to expose or have available any **implementation** detail of the component. Flows can trace data, control, or mixed flow by connecting event and data ports.)

- **properties**: specifies properties of the component that apply to all instances of this component unless overwritten in implementations or extensions

- **extends**: is used where a type extends another type, as shown for the **thread** type data_management in Table 4-3

If there are no entries under the subclause reserved words **features**, **flows**, or **properties**, they may be omitted, or the reserved word statement **none** can be used to signify explicitly that there are no entries. For example, the reserved word subclause **flows** is omitted in the **thread** type declaration for data_management and **none** is used in the

other empty subclause cases in Table 4-3. The use of **none** explicitly designates that the subclause is empty. The use of **none** avoids the misinterpretation of a developer's accidental omission of a subclause declaration as intentional.

In Table 4-3, these declarations under the **features** subclause in the type declaration for `simple_speed_control` define ports for the type:

```
raw_speed: in data port speed_type;
toggle_mode: in event port;
throttle_cmd: out data port throttle_data;
```

Notice that there is one **in data port** declaration in the **features** section of the type `system_management`. The type declaration for `data_management` extends the type `system_management`. Within this type extension declaration, the **in data port** `in_data` declaration is completed by including **refined to** and adding the **data** type `speed_type` to the **port** declaration, and an **out data port** declaration is added.

A component type declaration is terminated by the reserved word **end** followed by the component's type **classifier** and a semicolon (;).

4.5 Implementation Declarations

A component **implementation** declaration structure (① and ②) and a sample declaration (③) are shown in Table 4-4. The basic form (①) declares a distinct **implementation**. The second form (②) includes the reserved word **extends,** indicating that the declared **implementation** extends another.

In the sample declaration (③ in Table 4-4), a **thread implementation** with the name `control_laws.control_input` is declared as an **implementation** of the type `control_laws`. The **implementation** name is formed using the type identifier followed by a specific identifier for the **implementation**. These are separated by a period (.). Within the `control_laws.control_input` declaration, a single **data** subcomponent is declared, the reserved word statement (**none**) is used for the **calls** subclause, and the other subclauses are omitted.

Table 4-4: Component Implementation Declarations

```
component_category implementation implementation_name          ①
   refines type
   subcomponents
   calls
   connections
   flows
   modes
   properties
end implementation_name ;
```

```
component_category implementation implementation_name          ②
   extends another_implementation_name
   refines type
   subcomponents
   calls
   connections
   flows
   modes
   properties
end implementation_name ;
```

```
thread control_laws                                            ③
end control_laws;

data static_data
end static_data;

thread implementation control_laws.control_input
subcomponents
configuration_data: data static_data;
calls none;
end control_laws.control_input;
```

4.6 Package Declarations

Packages provide a way to organize component type declarations, **implementation** declarations, and **property** associations within an AADL specification. Each **package** introduces a distinct namespace for component **classifier** declarations, **port group** type declarations, **annex** library declarations, and **property** associations.

For example, a component type may be declared within a **package** and used in multiple subsystem declarations. This is shown in Table 4-5 where the **package** acutators_sensors includes a **device** speed_sensor that is used in the primary and backup **implementation** of the **system** control. Note that the **package** name with a double colon (**::**) is used to precede the **device** speed_sensor when it is referenced (e.g., in the subcomponent declaration within the **implementation** declarations). The comment line (-- ...) is used to indicate other declarations that are not shown. Packages are discussed in more detail in Section 12.1 (Packages).

Table 4-5: Example Packages

```
package actuators_sensors
public
device speed_sensor
end speed_sensor;
-- …
end actuators_sensors;

system control
end control;

system implementation control.primary
subcomponents
speed_sensor: device actuators_sensors::speed_sensor;
-- …
end control.primary;

system implementation control.backup
subcomponents
speed_sensor: device actuators_sensors::speed_sensor;
-- …
end control.backup;
```

4.7   Property Set Declarations

Property set declarations allow the addition of **properties** to the core AADL
property set. These additions can be used to support specialized modeling and analysis
capabilities that can be defined in AADL annexes. Declarations in an AADL specification can
refer to **packages** and property sets that may be separately stored. More details on
property set declarations can be found in Section 11.3 (Defining New Properties).
References to **property** names, types, and constants declared within a **property set**
are prefaced by the name of the **property set**.

4.8   Annex Library Declarations

Annex library declarations enable extensions to the core language concepts and syntax.
Often these extensions support custom analyses using specialized models and abstractions
(e.g., an error annex that supports reliability analysis). **Annex** libraries define a sublanguage
that can be used in **annex** subclauses within component type and **implementation**
declarations. **Annex** libraries are declared within packages and **annex** subclauses can be
included within component type and **implementation** declarations. These subclauses use
the elements declared in the **annex** library (e.g., associating values or expressing assertions
with elements of the **annex**).[11]

---

[11] The language can also be extended through *annex documents,* which are approved extensions to the
core AADL standard.

4.9   Namespaces

There is a global namespace for an AADL specification. **Packages** and **property set** names are in the global namespace. Their content can be named anywhere by preceding it with the **package** name. Component declarations placed directly in an AADL specification are visible only within that AADL specification. They are not accessible from within packages or other AADL specifications; they are considered to reside in an anonymous namespace. An AADL specification acts as a local work area whose component declarations are only locally visible.

4.10 Partial Specifications

A number of declarations within a syntactically and semantically correct specification can be partially completed. For example, neither the identity (**type** or **implementation**) of a component contained within another nor the data type for the ports in a data connection between components needs to be specified until a complete representation is instantiated from the specification (i.e., the design is finalized).

The flexibility to develop partial specifications can be used effectively during design, especially in the early stages where details may not be known or decided upon. This flexibility allows the syntactic checking of an incomplete specification and enables extended semantic, domain, or project-specific analysis to be conducted. For example, the detailed signal timing can be specified and signal latency can be analyzed without a complete or detailed specification of the representation of data communicated through ports or other elements of the design. Similarly, using the flow specification construct, end-to-end flows can be analyzed without the system hierarchy being detailed to the level required for instantiation.

4.11 Extends, Refines, and Partial Specification

When coupled with the **extends**, **refines**, and **implementation** facilities of the language, partial specification can be used to define a core type or **implementation** pattern. This core pattern can be used to generate a family of components (i.e., core patterns with less detail and descendants with more specific and modified declarations). Table 4-6 shows an example of the use of **extends**. The basic **system** component type declaration forms the core for two type extensions, basic_plus and control. Within the extensions, the data input **port** declaration input_data is completed with a **data** type, and an additional **port** is added.

A more detailed discussion of the extension and refinement capabilities and additional example patterns is presented in Section 12.2 (Design Patterns).

Table 4-6: A Simple Extends and Refines Example

```
system basic
features
input_data: in data port;
-- …
end basic;
--
system basic_plus extends basic
features
input_data: refined to in data port sensor_data;
in_event: in event port;
-- …
end basic_plus;
--
system control extends basic
features
input_data: refined to in data port speed_data;
in_event_data: in event data port;
-- …
end control;
--
data sensor_data
end sensor_data;
--
data speed_data
end speed_data;
```

5   Software Components

Software component abstractions represent processed source text (executable binary images) and execution paths through executable code. Executable binary images (i.e., executable code and data) are the result of processing (such as compiling or linking) source text associated with a component. A component's source text may be written in a conventional programming language (e.g., Ada, Java, or C), domain-specific modeling language (e.g., MatLab/Simulink), or hardware description language (e.g., VHDL). The source text may also be an intermediate product of processing those representations (e.g., an object file).

The AADL software component abstractions are

- process (Section 5.1): represents a protected address space
- thread (Section 5.2): represents a unit of concurrent execution
- thread group (Section 5.3): represents a compositional unit for organizing threads
- data (Section 5.4): represents data types and static data in source text
- subprogram (Section 5.5): represents callable sequentially executable code

5.1   Process

The **process** abstraction represents a protected address space, a space partitioning where protection is provided from other components accessing anything inside the **process**. The address space contains

- executable binary images (executable code and data) directly associated with the **process**
- executable binary images associated with subcomponents of the **process**
- server subprograms (executable code) and data that are referenced by external components

A **process** does not have an implicit **thread**. Therefore, to represent an actively executing component, a **process** must contain a **thread**.

5.1.1   Textual Representation

Table 5-1 contains a partial listing of the textual specification for a **process**. The **process** is shown with examples of all three of its allowed subcomponent categories: (1) **thread**, (2) **thread group**, and (3) **data**. In this listing, simplified type and **implementation** declarations for the components are provided. Two ports are shown, one as input and one as output for the **process**. In a complete specification, **connections** that define the

information flow would be declared within the **process implementation**. Only the subcomponent declarations of the **process implementation** of `control_processing.speed_control` are shown explicitly. Other details of the specification are not included. These omissions are legal for a syntactically correct partial specification as discussed in Section 4.10 (Partial Specifications).

Table 5-1: Textual Representation of a Sample Process

```
process control_processing
features
input: in data port;
output: out data port;
end control_processing;

process implementation control_processing.speed_control
subcomponents
control_input: thread control_in.input_processing_01;
control_output: thread control_out.output_processing_01;
control_thread_group: thread group
control_threads.control_thread_set_01;
set_point_data: data set_point_data_type;
end control_processing.speed_control;


thread control_in
end control_in;

thread implementation control_in.input_processing_01
end control_in.input_processing_01;

thread control_out
end control_out;

thread implementation control_out.output_processing_01
end control_out.output_processing_01;

thread group control_threads
end control_threads;

thread group implementation control_threads.control_thread_set_01
end control_threads.control_thread_set_01;

data set_point_data_type
end setpoint_data_type;
```

5.1.2 Graphical Repesenation

A graphical representation of the **process implementation** from Table 5-1 `control_processing.speed_control` is shown in Figure 5-1. The **process** is shown with examples of its allowed subcomponent categories: **thread**, **thread group**, and **data**. As shown in Figure 5-1, two threads (`control_input` and `control_output`), a single **data** component (`set_point_data`), and a **thread group** (`control_thread_group`) are contained within the **process implementation** `control_processing.speed_control`.
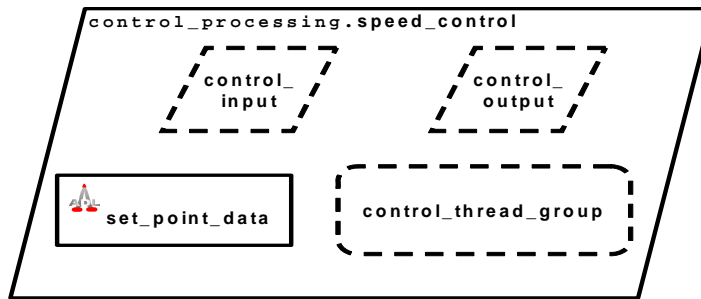
Figure 5-1:   *Graphical Representation of a Sample Process*

5.1.3   Properties

For the **process** and its subcomponent threads, predeclared **properties** for processes enable the specification of the

- runtime enforcement of memory protection
- relevant source file information
- source file loading times
- scheduling protocols
- binding constraints

In addition, there are **properties** that can be inherited and shared by a process's subcomponent threads (e.g., Period, Deadline, or Actual_Processor_Binding). These include predeclared **properties** as well as new **properties**, defined as prescribed in Section 11.3 (Defining New Properties).[12]

5.1.4   Constraints

An AADL **process** represents only a protected address space. Consequently, processes must contain at least one explicitly declared **thread** or **thread group** subcomponent. In other words, it is not equivalent to a POSIX process that represents both a protected address space and an implicit **thread**.

Table 5-2 summarizes the permitted type declaration and **implementation** declaration elements of a **process**. A **process** can only be a subcomponent of a **system** component. A summary of the allowed subcomponent relationships and features is included in the Appendix on pages 117–119.

---

[12]   There is a standard predeclared property set named AADL_Properties that is a part of every AADL specification [SAE 06a].

Table 5-2: Summary of Permitted Process Declarations

Category	Type	Implementation
process	Features: • server subprogram • port • provides data access • requires data access Flow specifications: yes Properties yes	Subcomponents: • data • thread • thread group Subprogram calls: no Connections: yes Flows: yes Modes: yes Properties yes

5.2   Thread

A **thread** is a concurrent schedulable unit of sequential execution through source code. Multiple threads represent concurrent paths of execution. A variety of execution **properties** can be assigned to threads, including timing (e.g., worst case execution times), dispatch protocols (e.g., periodic, aperiodic, etc.), memory size, and processor binding.

5.2.1   Textual Representation

Sample **thread** type, **implementation**, and **subcomponents** declarations are shown in Table 5-3. In Table 5-3, there are two **thread** type and three **thread implementation** declarations. Two of the **thread implementation** declarations describe separate implementations of the same **thread** type data_input. Instances of threads are defined in **subcomponents** subclause declarations of the **process implementation** data_management.

Related type and **implementation** declarations are grouped together in this example. This grouping of declarations is used for clarity and is not a required organization within a specification.

Table 5-3: A Sample Thread Declaration

```
thread data_processing
end data_processing;

thread implementation data_processing.integrated_data_processing
end data_processing.integrated_data_processing;

thread data_input
end data_input;

thread implementation data_input.roll_data_input
end data_input.roll_data_input;

thread implementation data_input.pitch_data_input
end data_input.pitch_data_input;

process data_management
end data_management;

process implementation
data_management.autonomous_submarine_data_management
subcomponents
roll_input: thread data_input.roll_data_input;
pitch_input: thread data_input.pitch_data_input;
attitude_data_processing: thread
data_processing.integrated_data_processing;
end data_management.autonomous_submarine_data_management;
```
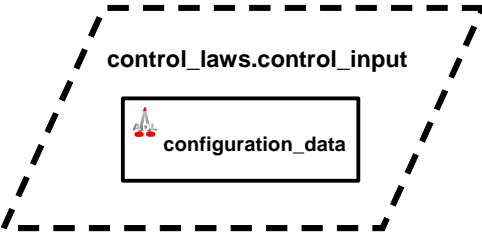
5.2.2   Graphical Representation

A graphical representation of the **thread implementation**
`control_laws.control_input` and its associated textual representation are shown in
Table 5-4. No interfaces for the type or other details of the type or **implementation**
declarations are shown.

In the example, the **data** instance `configuration_data` is defined as a subcomponent
of the **thread**, and the referenced identifier is a **data** type rather than a **data
implementation**. This is legal only if there are no **implementation** declarations of
the **data** type anywhere within the specification.

Table 5-4: A Sample Thread Implementation with One Subcomponent

control_laws.control_input configuration_data	``` thread control_laws end control_laws; data static_data end static_data; thread implementation control_laws.control_input subcomponents configuration_data: data static_data; end control_laws.control_input; ```

5.2.3 Thread Execution

A graphical state machine representation of **thread** execution is shown in Figure 5-2. A round-cornered rectangle represents an execution state of a **thread** or a composite state that includes at least one execution state. The ovals are non-execution states. Transitions between states are represented by directed arcs. Arcs may originate, join, diverge, or terminate at junction points depicted as small circles.

Instances of a **thread** can transition between various scheduling states as the result of normal execution (e.g., preemption or completion of initialization) or faults/errors. There are predefined entry points for each of the **thread** execution states: *Initialize*, *Compute*, and *Recover*. The initialize and compute entry points are used for normal execution.

If **thread** execution results in a fault that is detected, the source text may handle the error. If the error is not handled in the source text, the **thread** is requested to recover and prepare for the next dispatch. If an error is considered unrecoverable, its occurrence is propagated as an **event** through the thread's predeclared **out event data port** *Error* (not shown in Figure 5-2). All threads have an *Error* **out event data port** that allows an unrecoverable error with descriptive information to be signaled.
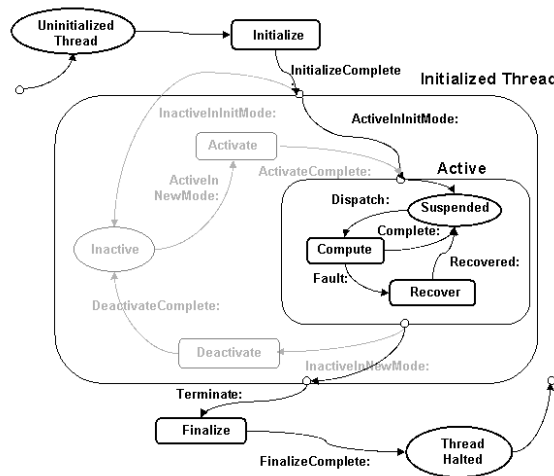
Figure 5-2:   Thread Execution State Machine

5.2.4 Properties

Predeclared **properties** support the detailed description of each of the execution phases of a **thread**. There are entry point **properties** that specify entry into code associated with each of these execution phases (Figure 5-2):

1. Initialize allows threads to perform application specific initialization.

2. Activate allows actions to restore application states between mode switches.

3. Compute represents the code to be executed on every thread dispatch.

4. Recover allows threads to perform fault recovery actions.

5. Deactivate allows actions to save application states between mode switches.

6. Finalize executes when thread is asked to terminate as part of a process unload or stop.

In addition, there are execution time and deadline **properties** for each of these execution phases (not shown in Figure 5-2).

A thread's scheduling-related **properties** include Dispatch_Protocol and Period. The supported protocols are

- periodic: repeated dispatches occurring at a specified time interval (a Period)

- aperiodic: event-triggered dispatch of threads

- sporadic: event-driven dispatch of threads with a minimum dispatch separation

- background: a dispatch initiated once with execution until completion

Periodic, aperiodic, and sporadic protocols typically involve hard deadlines for the **thread**. The predeclared and user-defined **thread properties** can be used to specify critical runtime aspects of a **thread** within a system's architectural representation, enabling the early analyses of **thread** behavior.

Table 5-5 is an example of some **property** associations for a **thread**. Entry points and associated execution times are declared for initialization and nominal execution.

Table 5-5: Sample Thread Properties

```
thread control
properties
-- nominal execution properties
Compute_Entrypoint => "control_ep";
Compute_Execution_Time => 5 ms .. 10 ms;
Compute_Deadline => 20 ms;
Dispatch_Protocol => Periodic;
-- initialization execution properties
Initialize_Entrypoint => "init_control";
Initialize_Execution_Time => 2 ms .. 5 ms;
Initialize_Deadline => 10 ms;
end control;
```

5.2.5   Constraints

Table 5-6 summarizes the legal subclause declarations for a **thread**.

Table 5-6: Summary of Permitted Thread Subclause Declarations

Category	Type	Implementation
thread	Features: • server subprogram • port • provides data access • requires data access Flow specifications: yes Properties yes	Subcomponents: • data Subprogram calls: yes Connections: yes Flows: yes Modes: yes Properties yes

A **thread** executes within the protected virtual address space of a **process**, either as an explicitly declared subcomponent or as a subcomponent of a **thread group** within a **process**. Thus, threads must be contained within (i.e., only be a subcomponent of) a **process** or a **thread group**. Multiple concurrent threads can exist within a **process**.

A summary of the allowed subcomponent relationships and features is included on pages 117–119 in the Appendix.

5.3 Thread Group

A **thread group** is a component abstraction for logically organizing **thread**, **data**, and **thread group** components within a **process**. Thread groups do not represent a virtual address space or a unit of execution. They provide a foundation for the separation of concerns in the design, defining a single reference to multiple threads and associated data (e.g., threads with a common execution rate or all threads and data components needed for processing input signals).

5.3.1 Textual Representation

Table 5-7 is a sample textual specification for a **thread group** that contains a **thread** component, two **data** components, and another **thread group**. Simplified **thread group** type and **implementation** declarations are shown. For example, only the **subcomponents** declarations part of the control.roll_axis component **implementation** declaration is shown. No details of the **thread group** **implementation** control_laws.roll are shown. Notice that the **data** subcomponent declarations control_data and error_data reference **data** **implementation** declarations rather than **data** type declarations, reflecting the flexibility that static **data** components can be declared at any level of the hierarchy. The **thread group** type declaration for control includes a **property** association that defines a Period of 50 ms. This value is assigned to (inherited by) all of the threads contained in the **thread group**.

Table 5-7: A Sample Thread Group AADL Textual Specification

```
thread group control
properties
Period => 50 ms;
end control;
--
thread group implementation control.roll_axis
subcomponents
control_group: thread group control_laws.roll;
control_data: data data_control.primary;
error_data: data data_error.log;
error_detection: thread monitor.impl;
end control.roll_axis;
--
thread monitor
end monitor;
--
thread implementation monitor.impl
end monitor.impl;
--
data data_control
end data_control;
--
data implementation data_control.primary
```

Table 5-7: A Sample Thread Group AADL Textual Specification

```
end data_control.primary;
--
data data_error
end data_error;
--
data implementation data_error.log
end data_error.log;
--
thread group control_laws
end control_laws;
--
thread group implementation control_laws.roll
end control_laws.roll;
```

5.3.2   Graphical Representation

Figure 5-3 contains a graphical representation of the **implementation** of the **thread group** control.roll_axis shown textually in Table 5-7. Notice that the names (identifiers) of the graphical subcomponents of the **thread group** match those contained in the textual representation of the thread group's **implementation** declaration. Partial declarations are permitted in the initial specification of the system (e.g., subcomponent declarations may not have component type or **implementation** references). This partial specification capability is particularly useful during early design stages where details may not be known or decided. Component **classifier** references can be added or completed in subcomponent refinements or declared in component **implementation** extensions. For example, in Table 5-7 the declaration for the subcomponent error_detection does not have to include the **thread** component **classifier** monitor.impl. This reference could be added later in an extension of the **thread group implementation** control.roll_axis.
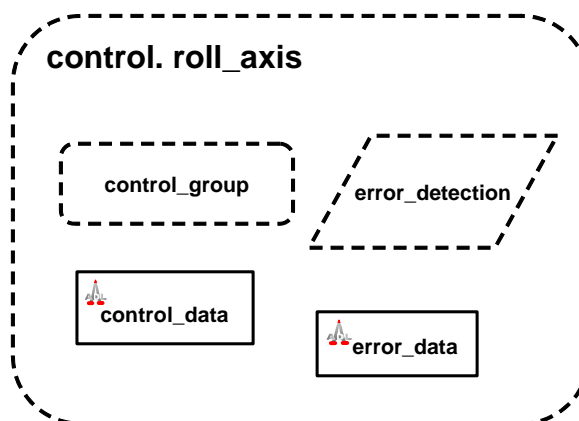
Figure 5-3:   A Sample Thread Group Graphical Representation

5.3.3 Properties

Predeclared **thread group properties** include declarations relating to the specification of

- source text
- timing characteristics
- relevant memory, processor, and connection bindings[13]

For example, there are Actual and Allowed_Processor_Binding **properties** for threads within the **thread group**, as well as **properties** that describe **thread** handling during **mode** changes (e.g., Active_Thread_Handling_Protocol that specifies the protocol to use for execution at the time instant of an actual mode switch).[14]

5.3.4 Constraints

A **thread group** can be a subcomponent only of a **process** or another **thread group**. Table 5-8 summarizes the permitted elements of a thread group's type and **implementation** declarations.

Table 5-8: Elements of a Thread Group Component

Category	Type	Implementation
thread group	Features: • server subprogram • port • provides data access • requires data access Flow specifications: yes Properties yes	Subcomponents: • data • thread • thread group Subprogram calls: no Connections: yes Flows: yes Modes: yes Properties yes

A summary of the allowed subcomponent relationships and features is included on pages 117–119 in the Appendix.

---

[13] The mapping of software to hardware components of a system that are required to produce a physical system implementation is called binding [SAE 06a].

[14] Actual_Processor_Binding, Allowed_Processor_Binding, and Active_Thread_Handling_Protocol are predeclared properties in the standard predeclared property set AADL_Properties.

5.4 Data

The AADL **data** abstraction represents static data (e.g., numerical data or source text) and data types within a system. Specifically, data component declarations are used to represent

- application data types (e.g., used as data types on ports and parameters)
- the substructure of data types via data subcomponents within data implementation declarations
- data instances

Data types in the application system can be modeled by **data** component type and **implementation** declarations. A **data** type (and **implementation**) declaration can be used to define the data associated with ports and parameters. It is sufficient to model an application source text data type with a **data** component type and relevant **property** declarations; it is not necessary to declare a **data implementation**. Consistency checks can be done on the **data** type associated with connections between ports and parameters. **Data** subcomponent declarations can be used to define the substructure of **data** types and instances. For example, fields of a record can be declared as **data** subcomponents in a **data implementation** declaration.

Data instances are represented by **data** subcomponent declarations within a software component or **system implementation**. Currently **data** subcomponents cannot be declared in subprograms. For example, data instances within source text (e.g., the instance variables of a class or the fields of a record) are represented by **data** subcomponent declarations in a **data** component **implementation**. These **data** instances can be declared as accessible by multiple components as outlined in Section 8.3 (Subcomponent Access). **Data** component types can have subprograms as **features**, allowing for modeling of abstract data types or classes with **access** methods.

5.4.1 Textual Representation

Sample **data** type and **implementation** declarations are shown in Table 5-9 that includes three **data** type declarations and a **data implementation** declaration address.others of the **data** type declaration address. In addition, a **thread implementation** declaration is shown with **data** subcomponents that reference the **data** types defined in Table 5-9.

As the commented description in the table explains, the first part of the table shows the **data** type string used in a **port** declaration. Specifically, it shows the declaration of a **data** type speed_data_type used to declare the **data** type for an input **data port** of the **process** controller. The **property** association defines the size of the **data** type as 16 bits. Only relevant portions of the controller **process** type declaration are included. The second part of the table shows an example of the declaration of the substructure of a **data implementation**. The substructure of the **data implementation**

address.others consists of four **data** subcomponents with **data** types string and int. In the third and final portion of the table, the **thread implementation** declaration for address_processing.address_lookup includes a specific **data** instance of the **data implementation** address.others as a subcomponent.

Notice that the **data** subcomponent declarations within the **data implementation** address_others reference only the **data** type declaration. **Subcomponents** subclauses can reference a **data** type declaration rather than a **data implementation** declaration only if there is no more than one **implementation** of that **data** type.

Table 5-9: Sample Data Component Declarations

```
-- string as a data type used in a port declaration --
data speed_data_type
properties
Source_Data_Size => 16 bits;
end speed_data_type;
--
process controller
features
input: in data port speed_data_type;
end controller;
--
-- a data implementation with substructure
data address
end address;
--
data implementation address.others
subcomponents
street : data string;
streetnumber: data int;
city: data string;
zipcode: data int;
end address.others;
--
-- supporting data declarations
data string
end string;
--
-- int as type
data int
properties
Source_Data_Size => 64b;
end int;
--
-- a data instance of the data implementation "address.others"
thread address_processing
end address_processing;
--
thread implementation address_processing.address_lookup
subcomponents
address_01: data address.others;
end address_processing.address_lookup;
```

5.4.2 Graphical Representation

Figure 5-4 contains graphical and corresponding textual representations for the **data** subcomponents of the **data implementation** address.others and the **thread implementation** address_processing.address_lookup presented in

Table 5-9.
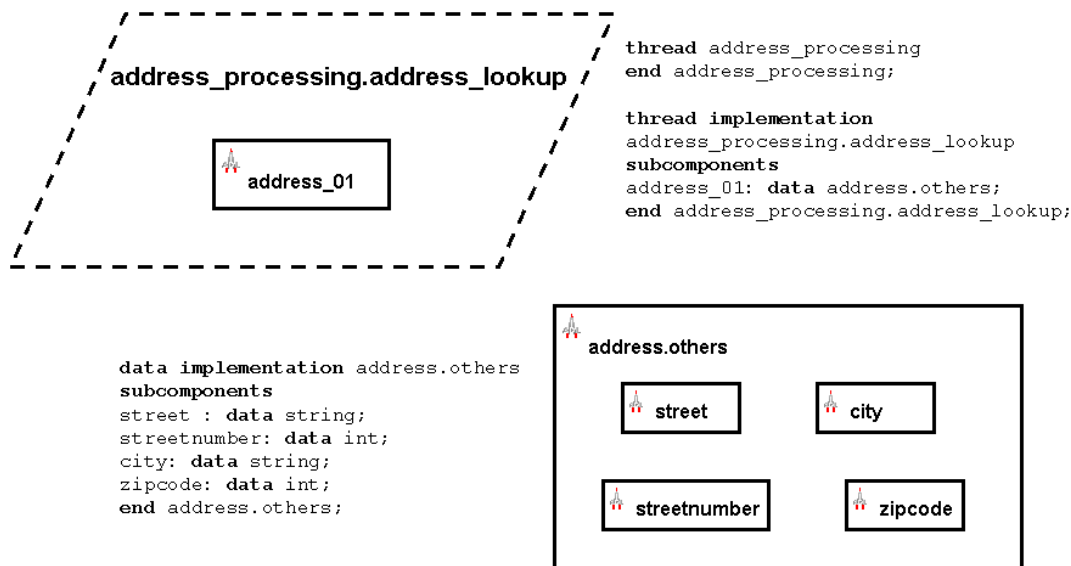
Figure 5-4:   Sample Data Component Graphical Representations

5.4.3 Properties

The predeclared **properties** for **data** components enable specification of

- source text for the **data** component
- name of the relevant **data** type declaration
- name of the relevant static data variable in the source text
- data size
- concurrency **access** protocol for shared data

Base types can be modeled using **data** types by

1. defining a new **property** (such as BaseType) that takes a (**data**) **classifier** as **property** value
2. applying this **property** to **data** components
3. declaring **data** component base types (such as SignedInt16 or UnsignedInt8)

For example, `BaseType =>` **`classifier`** `BaseTypes::SignedInt16`; could be a **`property`** declared in the **`data`** type `speed_data_type`, where the **`data`** type `SignedInt16` is declared in the **`package`** `BaseTypes`.

5.4.4 Constraints

Table 5-10 summarizes the legal elements within **`data`** type and **`data implementation`** declarations. Notice that only **`data`** components can be subcomponents within a **`data`** component.

A **`data`** component can be a subcomponent of a **`data`**, **`thread`**, **`thread group`**, **`process`**, or **`system`** component. A summary of the allowed subcomponent relationships and features is included on pages 117–119 in the Appendix.

Table 5-10: Legal Elements of Data Type and Implementation Declarations

Category	Type	Implementation
data	Features: • subprogram • provides data access Flow specifications: no Properties yes	Subcomponents: • data Subprogram calls: no Connections: access Flows: no Modes: yes Properties yes

A **`data`** subcomponent subclause can reference a **`data`** type declaration that does not have a **`data implementation`**. For example, the reference for the subcomponent `street` of the **`data implementation`** `address.others` shown in Figure 5-4 is to the **`data`** type `string`. However, if a **`data`** type declaration has more than one associated **`data implementation`** declaration, both the component type and a component **`implementation`** must be present in a component **`classifier`** reference in order to completely identify the **`classifier`**.

5.5 Subprogram

The **`subprogram`** component abstraction represents sequentially executable source text—a callable component with or without parameters that operates on data or provides server functions to components that call it. A **`subprogram`** and its **`parameter`** signature are declared through component declarations but are not instantiated as subcomponents. Instead, **`calls`** to subprograms are declared in **`calls`** sequences in **`thread`** and **`subprogram`** implementations. More details on **`calls`** to subprograms and example **`calls`** declarations are provided in Section 8.4 (Subprogram Calls).

The modeling roles for subprograms include the representation of

- a method call for operation on **data**
- basic program **calls** and call sequencing
- remote service/procedure **calls**

These calls can include data transfer into or out of the **subprogram**. Parameters, declared as **features** of a **subprogram**, provide the interface for the transfer of data into or out of a **subprogram**.

5.5.1 Textual Representation

Table 5-11 is an example of a **subprogram** representing a service (method) call for operation on **data**. It shows the relevant component type and **implementation** declarations and the declaration of that **subprogram** as one of the **features** scale_acc_data within a **data** component accelerometer_data. The **feature** scale_acc_data represents an entry point into source text that operates on the **data** component accelerometer_data.

Table 5-11:   Subprogram Textual Representation

```
subprogram scale_data
end scale_data;
subprogram implementation scale_data.scale_sensor_data
end scale_data.scale_sensor_data;
data accelerometer_data
features
scale_acc_data: subprogram scale_data.scale_sensor_data;
end accelerometer_data;
process sensor_systems
end sensor_systems;
process implementation sensor_systems.sensor_processing
subcomponents
acc_data: data accelerometer_data;
scale_it: thread process_data.scale;
end sensor_systems.sensor_processing;
```

5.5.2 Graphical Representation

Figure 5-5 contains graphical and corresponding textual representations for the **process implementation** sensor_systems.sensor_processing shown in Table 5-11. The **subprogram** scale_acc_data is represented by an oval that adorns the **data** subcomponent acc_data of the **process implementation** sensor_systems.sensor_processing. The **thread** scale_it is not shown in the figure.
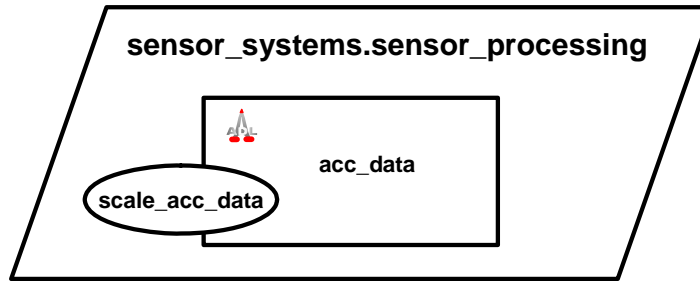
Figure 5-5:Subprogram Graphical Representation

Table 5-12 shows both textual (upper portion) and graphical (lower portion) representations of an example of a **subprogram** abstraction representing a **server subprogram**.

In this textual representation, the two **process implementation** declarations (`control.temp_control` and `manage_data.manage_temp`) are bound to separate **memory** components (e.g., memories associated with individual processing nodes on a distributed computing network). The **thread implementation** `control_law.linear` within the `control.temp_control` **process implementation calls** the **subprogram** `acquire.temp` that is declared as a **server subprogram** feature in the **thread** type `read`.

In the graphical representation of the specification shown in the lower portion of Table 5-12, the subroutine entry point `read_it` is identified as a feature of the subcomponent **thread** `temp_reader`. In addition, the call `get_temp` is shown in the **thread** `control.temp_control`, and the **binding** of this call to the `read_it` **subprogram** is shown with an arrowed line. This call can be a remote call, where the **server subprogram thread** `temp_reader` is bound to a separate **processor** than the calling **thread** `linear01`. More details on **subprogram calls** and a remote client-server example can be found in Section 8.4 (Subprogram Calls).

Table 5-12:   Example Textual and Graphical Subroutine Declarations

```
process control
end control;
--
process implementation control.temp_control
subcomponents
linear01: thread control_law.linear;
end control.temp_control;
--
thread control_law
end control_law;
--
thread implementation control_law.linear
calls {
      get_temp: subprogram acquire.temp; };
end control_law.linear;
```

Table 5-12:   Example Textual and Graphical Subroutine Declarations (cont.)

```
process manage_data
end manage_data;
--
process implementation manage_data.manage_temp
subcomponents
temp_reader: thread read.read_temp;
end manage_data.manage_temp;
--
thread read
features
read_it: server subprogram acquire.temp;
end read;
--
thread implementation read.read_temp
end read.read_temp;
--
subprogram acquire
end acquire;
--
subprogram implementation acquire.temp
end acquire.temp;
```
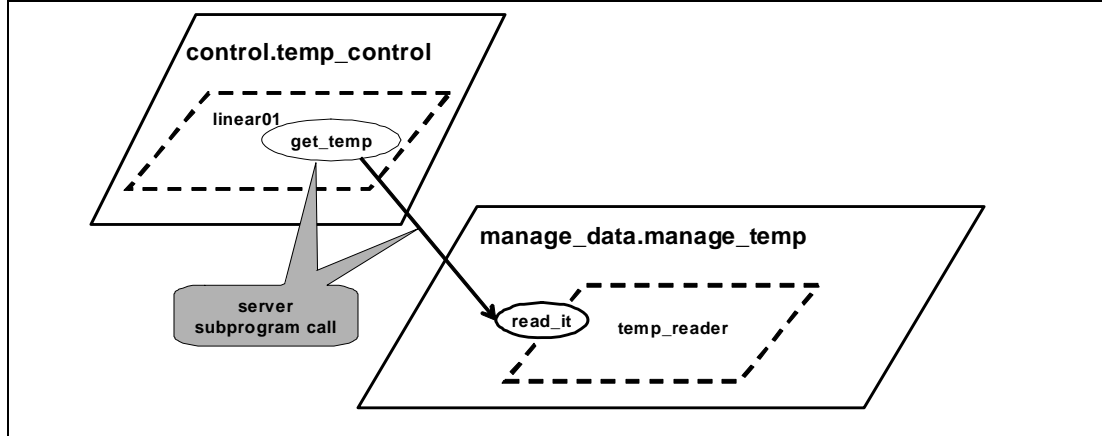

5.5.3   Properties

Predeclared **subprogram properties** include declarations relating to the

- source text for the **subprogram**

- **memory** requirements

- **memory binding**

- characteristics related to **calls** into the **subprogram**

5.5.4 Constraints

Table 5-13 summarizes the permitted elements of a subprogram's component type and **implementation** declarations.

Table 5-13:   Restrictions on Subprogram Declarations

Category	Type	Implementation
subprogram	Features: • out event port • out event data port • port group • requires data access • parameter Flow specifications: yes Properties yes	Subcomponents: • none Subprogram calls: yes Connections: yes Flows: yes Modes: yes Properties yes

The interactions of subprograms are constrained to

- **event**-based interfaces: **out event port**, **out event data port**, and a **port group** consisting only of these event port types

- **data** interfaces: through parameters of **calls** to and returns from the **subprogram**

Out event ports and **out event data** ports support modeling subprograms that raise an **event** (with or without associated data) that must be passed through an enclosing **thread** to other components. A **subprogram** may require access to data but cannot contain static **data** subcomponents.

6 Execution Platform Components

Execution platform components represent computational and interfacing resources within a system. This representation includes complex hardware and associated software systems. For example, in one model a Linux computing resource can be represented as a **processor** and, in an **implementation** model of the **processor**, as a **system** with Linux software mapped onto an execution platform **processor**.

There are four categories of execution platform components in the AADL:

1. processor (Section 6.1): represents components that execute threads

2. memory (Section 6.2): represents components that store data and code

3. bus (Section 6.3): represents components that provide access among execution platform components

4. device (Section 6.4): represents components that interface to the external environment

Within an AADL specification, software components must be mapped onto execution platforms through **binding** relationships. These bindings define where code is executed and data and executable code are stored within a system. For example, a **thread** must be bound to a **processor** for execution and a **process** must be bound to **memory**. Similarly, connections among components within a system must be bound to appropriate execution platform components (e.g., a simple connection is bound to a single **bus** or a connection within a complex distributed system is bound to a sequence of buses and intermediate processors and devices). Additional information on **binding** is in Section 7 (System Structure and Instantiation).

A collection of execution platform components contained within an AADL system abstraction can be used to model complex physical computational resources. For example, **memory** that represents a hard disk and a processor that supports software execution within a system can model a database server. Similarly, a collection of software and execution platform components (i.e., a **system implementation**) can represent a virtual machine layer within a layered system architecture model.

6.1 Processor
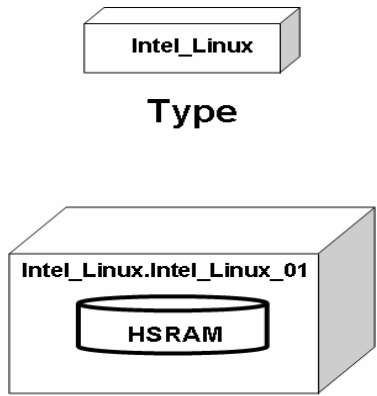
A **processor** is an abstraction of hardware and associated software that is responsible for scheduling and executing threads. Processors can execute threads that are declared in application software systems or threads that reside in components accessible from those processors.

Processors themselves may have embedded software (e.g., an operating system) that implements scheduling and other capabilities that support **thread** execution. Alternatively, separate software components or other software virtual machines can supply this support, provided that software is bound to **memory** that is accessible by the **processor**.

6.1.1   Textual and Graphical Representations

Table 6-1 shows a type and **implementation** declaration for a **processor**. Both textual and corresponding graphical representations are shown. In this example, a single **processor system** with **memory** contained inside of the **processor** is shown. No other interconnections are required.

Table 6-1: A Sample Processor Textual and Graphical Representation


```
processor Intel_Linux
properties
Hardware_Source_Language=> VHDL;
Hardware_Description_Source_Text  =>
"intel_vhdl_1, intel_vhdl_2";
end Intel_Linux;
--
processor implementation
Intel_Linux.Intel_Linux_01
subcomponents
HSRAM: memory RAM.Intel_RAM;
end Intel_Linux.Intel_Linux_01;
--
memory RAM
end RAM;
--
memory implementation RAM.Intel_RAM
end RAM.Intel_RAM;
```

In the textual representation, the **properties** subclauses define the hardware description language (`Hardware_Source_Language`) and the files that contain the source text for the hardware description (`Hardware_Description_Source_Text`). The **processor implementation** declaration of `Intel_Linux.Intel_Linux_01` includes a single **memory** subcomponent `HSRAM`. The **memory** subcomponent's type and **implementation** declarations are shown.

The corresponding graphical representations of type and **implementation** are shown to the right of the textual representation in Table 6-1. The nesting of the **memory** graphic (labeled `HSRAM`) within the **processor** graphic shows containment. The optional bold line (discussed in Section 4.3) is not used for the **processor implementation** graphic.

6.1.2 Properties

Predeclared **processor properties** can be used in a **processor** declaration. In addition to the hardware description **properties** included in the example from Table 6-1, other **properties** include a Scheduling_Protocol **property** that must have a value if threads are bound to the **processor** and an Allowed_Dispatch_Protocol **property** that specifies the dispatch protocols supplied by the **processor**.[15]

6.1.3 Constraints

Table 6-2 summarizes the permitted elements of a processor's type and **implementation** declarations.

Table 6-2: Summary of Permitted Processor Declarations

Category	Type	Implementation
processor	Features: • server subprogram • port • port group • requires bus access Flow specifications: yes Properties yes	Subcomponents: • memory Subprogram calls: no Connections: no Flows: yes Modes: yes Properties yes

A **processor** can only be a subcomponent of a system component. A summary of the allowed subcomponent relationships and features is included on pages 117–119 in the Appendix.

6.2   Memory

Memory abstractions represent storage components for data and executable code (i.e., **subprograms**, **data**, and **processes** are bound to **memory** components). **Memory** components include randomly accessible physical storage (e.g., RAM, ROM) or complex permanent storage such as disks or reflective memory. Since they have a physical runtime presence, **memory** components have **properties** such as word size and word count.

The **memory** component can represent memory inside of a **processor** or a separate execution platform unit that must be connected to a processor through a **bus**. Memory banks can be modeled as a single or composite **memory** unit.

---
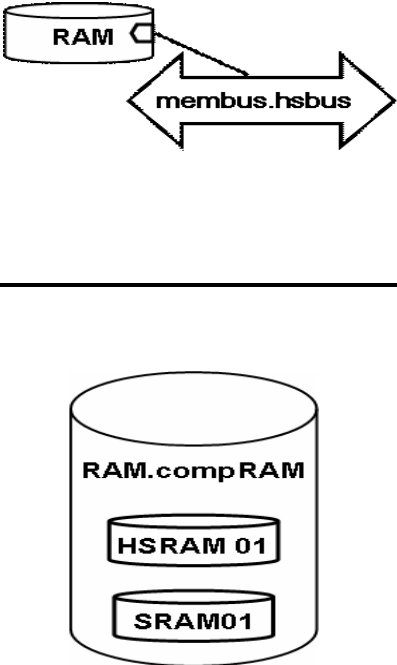
[15]   There is a standard predeclared property set named AADL_Properties that is a part of every AADL specification [SAE 06a].

6.2.1   Textual and Graphical Representations

An example **memory** declaration and its graphical representation are shown in Table 6-3. In this example, a **memory** of the type RAM is declared with a single **feature** bus01 that establishes that all instances of RAM require access to the **bus membus.hsbus**. No explicit **properties** for this type are declared. The type and **implementation** declarations for the **requires bus access to** bus01 are shown at the end of the listing.

The **memory implementation** RAM.compRAM declares that this **implementation** of the **memory** type RAM includes **memory** subcomponents HSRAM01 and SRAM01. No **modes** or **properties** are declared. The subcomponents of the **memory implementation** RAM.compRAM are declared as implementations of a common type XRAM. An expanded **memory** composition can be used to model a complicated memory bank. These examples show that **memory** can only contain other **memory** components and must be connected to a **bus** unless it is enclosed in a **processor**.

Table 6-3: A Sample Memory Textual and Graphical Representation

```
memory RAM
features
bus01: requires bus access membus.hsbus;
end RAM;
--
memory implementation RAM.compRAM
subcomponents
HSRAM01: memory XRAM.HSRAM;
SRAM01: memory XRAM.SRAM;
end RAM.compRAM;
--
memory XRAM
end XRAM;
--
memory implementation XRAM.HSRAM
end XRAM.HSRAM;
--
memory implementation XRAM.SRAM
end XRAM.SRAM;
--
bus membus
end membus;
--
bus implementation membus.hsbus
end membus.hsbus;
```

6.2.2   Properties

Predeclared **memory properties** include

- **memory** access protocol

- word size

- other important descriptive characteristics of storage units

- The default value for **memory** access (Memory_Protocol) is read–write but can be associated with the values of read only or write only.

6.2.3   Constraints

Table 6-4 lists the permitted elements of **memory** type and **implementation** declarations.

Table 6-4: Summary of Permitted Memory Declaration Subclauses

Category	Type	Implementation
memory	Features • requires bus access Flow specifications: no Properties yes	Subcomponents: • memory Subprogram calls: no Connections: no Flows: no Modes: yes Properties yes

A **memory** component can only be contained within a **memory**, **processor**, or **system** component. Moreover, an individual **memory** component must be contained in a **processor**, declared a subcomponent of a **memory** unit, or connected to a **processor** through a **bus**. A summary of the allowed subcomponent relationships and features is included on pages 117–119 in the Appendix.

6.3   Bus
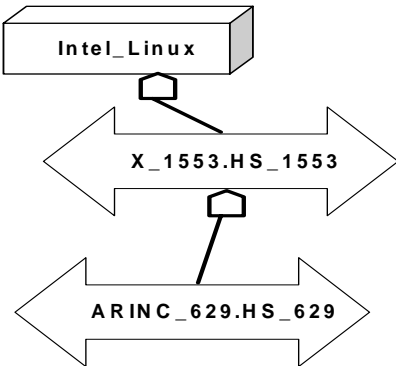
A **bus** represents hardware and associated communication protocols that enable interactions among other execution platform components (i.e., **memory**, **processor**, and **device**). For example, a connection between two **threads**, each executing on a separate **processor**, is over a **bus** between those processors. This communication is specified using **access** and **binding** declarations to a **bus**. Buses can be connected directly to other buses to represent complex inter-network communications. Thus, connections between components can be bound to a sequence of buses or a sequence of buses with intervening processors.

6.3.1 Textual and Graphical Representations

Since a **bus** acts only as a shared component, its interactions (**features**) are specified as **bus access features** in component type declarations. For example, a **processor** requires access to a **bus** in order to communicate with **memory** that contains the threads executing on that **processor**. Similarly, a **bus** may require access to another **bus**. Alternatively, a **system** may provide access to one of its **bus** subcomponents.

Table 6-5 shows a portion of an AADL textual specification and its corresponding graphical representation. Included in the example are a **processor** type declaration for Intel_Linux and two **bus** type declarations for X_1553 and ARINC_629. The **processor** type declaration for Intel_Linex includes a **requires bus access** declaration for the **bus** X_1553.HS_1553 and the **bus** type declaration X_1553 includes a **requires bus access** for the **bus** ARINC_629.HS_629. These required accesses are shown in the graphic on the right side of Table 6-5. The **implementation** declarations for both buses are also shown in the textual specification in Table 6-5.

Table 6-5: A Sample Bus Specification: Textual and Graphical Representation


```
processor Intel_Linux
features
A1553: requires bus access X_1553.HS_1553;
end Intel_Linux;
--
bus X_1553
features
A629: requires bus access
ARINC_629.HS_629;
end X_1553;
--
bus implementation X_1553.HS_1553
end X_1553.HS_1553;
--
bus ARINC_629
end ARINC_629;
--
bus implementation ARINC_629.HS_629
end ARINC_629.HS_629;
```

6.3.2 Properties

There are a number of predeclared **properties** that can be used to specify important **bus** characteristics:

- transmission characteristics such as allowed connection and access protocols, message sizes, transmission time, propagation delay
- hardware source language descriptions
- data movement time characteristics such as the time to move a byte or block of data and any fixed data movement overhead time

6.3.3 Constraints

Table 6-6 summarizes the permitted elements of **bus** type and **implementation** declarations.

Table 6-6: Summary of Permitted Bus Declaration Subclauses

Category	Type	Implementation
bus	Features • requires bus access Flow specifications: no Properties yes	Subcomponents: • None Subprogram calls: no Connections: no Flows: no Modes: yes Properties yes

A **bus** component can only be a subcomponent of a **system** component. A summary of the allowed subcomponent relationships and features is included on pages 117–119 in the Appendix.

6.4 Device

Device abstractions represent entities that interface with the external environment of an application system. Those devices often have complex behaviors. They may have internal processors, memory, and software that are not explicitly modeled. Alternatively, they may require driver software that is executed on an external processor. A device's external driver software may be considered part of a processor's execution overhead, or it may be treated as an explicitly declared **thread** with its own execution **properties**. Examples of devices are sensors and actuators or standalone systems such as a Global Positioning System.

6.4.1 Textual and Graphical Representations
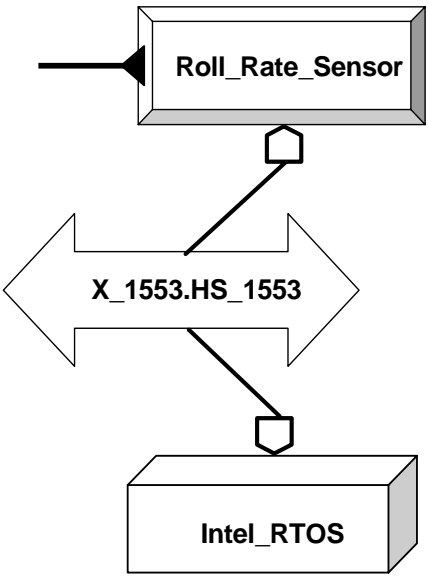
A **device** can interact in complex ways with other components. For example, a **device** may have a physical connection to a **processor** via a **bus** as well as logical connections through ports to application software components. As with all logical connections among components residing on distinct execution platform elements, these logical connections must be supported by (be bound to) physical connections.

Table 6-7 shows an excerpt from an AADL specification that describes a **device** Roll_Rate_Sensor interacting through a **bus** with a **processor** Intel_RTOS. The **processor** executes the device driver for the Roll_Rate_Sensor. The requirement for **bus access** is specified in the type declaration for Roll_Rate_Sensor. Similarly, the need for **bus access** is declared within the **processor** type declaration for Intel_RTOS. Notice that the **out data port** declared on the roll rate sensor **device** provides the rate data from the sensor. A **device** can be used to represent a more complex

physical element, such as an engine where the ports can represent the engine's sensors and actuators.

Table 6-7: A Sample Device Specification: Textual and Graphical Representation

```
processor Intel_RTOS
features
A1553: requires bus access
X_1553.HS_1553;
end Intel_RTOS;
--
device Roll_Rate_Sensor
features
A1553: requires bus access
X_1553.HS_1553;
raw_roll_rate: out data port;
end Roll_Rate_Sensor;
--
bus X_1553
end X_1553;
--
bus implementation X_1553.HS_1553
end X_1553.HS_1553;
```

Devices can be viewed from different perspectives. They are integral to the execution environment, both in terms of the application computing system (software and execution platform components) and the physical environment in which the application system exists. Thus, a **device** can be viewed as

- a physical component that interfaces with the application software through ports (and port groups), as shown in Figure 6-1

- part of the application system interacting with execution platform components and the application system, as shown in Figure 6-2

- a unit in the environment that is accessed or controlled by the application system, as shown in Figure 6-3

The complexity and nature of interactions of a **device** depend upon how it is included in the architecture. If a **device** is included as part of the execution platform system, there are numerous logical connections to the application system. If it is included as part of the application system, there are physical connections via **bus access** across the system hierarchy. In general, it is preferable to place the **device** declaration with the application code, since the emphasis is on its interaction with the application and the number of connections to the execution platform is then limited to the **bus**.
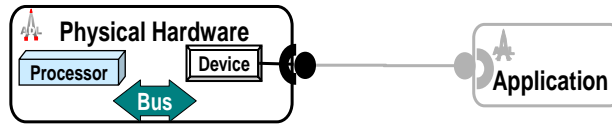
Figure 6-1:   A Device as Part of the Physical Hardware
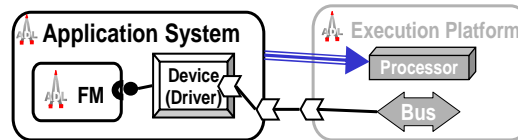
Figure 6-2:   A Device as Part of the Application System
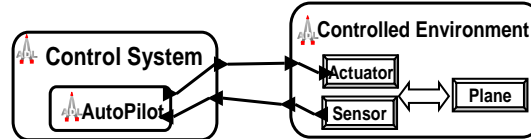
Figure 6-3:   A Device as Part of the Controlled Environment

The **data port**, **port group**, and **connections** abstractions—along with their graphical representations—depicted in Figure 6-1 through Figure 6-3 are discussed in Section 8: Component Interactions.

6.4.2   Properties

Device properties encompass the dual software and hardware character of a **device**.

- software-specific **properties**
  - source code files
  - source code language
  - code size
  - execution platform binding properties
- execution platform (hardware) **properties**, such as those specifying the files that contain the hardware description language for the device and the language used for that description
- **properties** for specification of the **thread** properties of the device software executing on a **processor**, such as dispatch protocols and execution time-related properties

6.4.3   Constraints

Table 6-8 summarizes the permitted elements of **device** type and **implementation** declarations. A **device** component can only be a subcomponent of a **system** component. A summary of the allowed subcomponent relationships and features is included on pages 117–119 in the Appendix.

Table 6-8: Summary of Permitted Device Declaration Subclauses

Category	Type	Implementation
device	Features • port • port group • server subprogram • requires bus access Flow specifications: yes Properties yes	Subcomponents: • none Subprogram calls: no Connections: no Flows: yes Modes: yes Properties: yes

7 System Structure and Instantiation

This section presents the language abstractions for structuring and integrating AADL elements into a complete representation of an application system that includes a system component, component bindings, source code elements, and instantiation.

7.1 System Abstraction

The **system** abstraction represents a composite of software, execution platform, or system components. **System** abstractions can be organized into a hierarchy that can represent complex systems of systems as well as the integrated software and hardware of a dedicated application system (e.g., flight navigation system or database server). Used early in the modeling process to generically represent a component, **system** components can be formed into a model that is transformed later—some system components being translated into **process** components and contained components being translated into **thread** and **thread group** components.

7.1.1 Textual and Graphical Representations

A **system** can consist of various combinations of software, execution platform, and system components. For example, a **system** may consist only of software (i.e., **process** or **data** components) or execution platform components. **Thread** and **thread group** components cannot be subcomponents of a **system**, since they must be contained within a **process** or a **thread group**.

The composition of a **system implementation** is declared through subcomponent declarations. Table 7-1 provides textual and graphical representations of a **system implementation** of the **system** type integrated_control. The details of the type declaration are not included. The explicit subcomponent declarations are shown in the **system implementation** declaration of integrated_control.integrated_control_system. However, many of the other subclauses are omitted. The supporting declarations are not shown (e.g., the **process** type declaration for the **process** type **controller**). In the graphical portrayal of the **system implementation**, the subcomponents of integrated_control_system of the type integrated_control are shown.

Table 7-1: A Sample System Specification: Textual and Graphical Representation

```
system integrated_control
end integrated_control;
--
system implementation integrated_control.integrated_control_system
subcomponents
control_process: process controller.speed_control;
set_point_data: data set_points;
navigation_system: system core_system.navigation;
real_time_processor: processor rt_fast.rt_processor;
hs_memory: memory rt_memory.high_speed;
high_speed_bus: bus network_bus.HSbus;
end integrated_control.integrated_control_system;
```

7.1.2 Constraints

Table 7-2 summarizes the permitted elements of a **system** type and **implementation** declarations. Notice that a system cannot contain a **thread** or **thread group**; they must be contained in a **process**. A **system** can be a subcomponent only of another system component. A summary of allowed subcomponent relationships and features is included on pages 117–119 in the Appendix.

Table 7-2: Summary of Permitted System Declarations

Category	Type	Implementation
system	Features: • server subprogram • port • port group • provides data access • provides bus access • requires data access • requires bus access Flow specifications: yes Properties yes	Subcomponents: • data • process • processor • memory • bus • device • system Subprogram calls: no Connections: yes Flows: yes Modes: yes Properties yes

7.2   System Instance

A **system** instance represents the runtime architecture of an operational physical system. That physical system may be a stand-alone system or a system of systems. A **system** instance consists of application software components and execution platform components. Component type and component **implementation** declarations are architecture blueprints that define the structure and connectivity of a physical system architecture. They must be instantiated to create a complete **system** instance. A **system** instance that represents the containment hierarchy of the physical system is created by instantiating a top-level **system implementation** and then recursively instantiating the subcomponents and their subcomponents.

Once instantiated, the application component instances can be bound to execution platform components (i.e., each **thread** is bound to a **processor**; each source text, **data** component, and **port** is bound to **memory** and each connection is bound to a **bus** if necessary). There is no explicit textual representation for **system** instances. Instead, **system** instances are created and stored as **system** instance models in XML. **System** instance models can be operated on by analysis and generation tools.

In a *fully specified system*, the application components are modeled to the level of threads and possibly refined to **subprogram calls** within threads. Similarly a fully specified execution platform includes processors to execute application code, memory to store

application code and data, devices that represent the physical environment of the embedded application, and buses that connect these components. Certain system analyses require fully specified system models. For example, scheduling analysis cannot be performed until all the application threads are specified and are bound to processors.

Early in the development process it is desirable to have *partially specified system* models and be able to instantiate them for analysis. For example, we may represent an application **system** as a collection of interacting subsystems without providing details of their implementation. Subsystems are modeled as **system** components or **process** components. We can instantiate this partial application **system** together with an execution platform model into a partial **system** instance model. We can assign resource budgets in terms of CPU cycles and memory requirements to the application subsystems and resource capacities to the execution platform. Given this data we can analyze various bindings of application components to the execution platform and ensure that the budgets do not exceed the capacity. We can also add **flow** specifications to individual subsystem components and end-to-end **flows** to the application system. Based on these **flow** specifications, flow analyses such as an end-to-end response time analysis can be performed without a fully detailed **system** model.[16]

7.3 Binding to Execution Platform Components

For a complete **system** specification (one that can be instantiated), software components must be bound to appropriate execution platform components. For example, **threads** must be bound to processing elements and **processes** must be bound to **memory**. Similarly, interprocessor connections must be bound to **buses**, and **subprogram calls** must be bound to their **server subprogram**. These bindings are defined through **property** associations.

There are three categories of **binding properties** that provide support for declaring:

1. allowed bindings
2. actual bindings
3. identified available memory and processor resources

For example, there is an Allowed_Memory_Binding predeclared **property** that identifies possible **memory** components for binding and an Actual_Memory_Binding predeclared **property** that specifies the **memory** component to which code and data from source text is bound. The Available_Memory_Binding **property** specifies the set of contained **memory** components that are available for the binding to a system's internal components from outside the system.[17]

---

[16] For more information on analysis, see AADL publications and presentations at www.aadl.info.

[17] Allowed_Memory_Binding and Actual_Memory_Binding are predeclared properties in the property set AADL_Properties that is part of every AADL specification [SAE 06a].

---

8 Component Interactions

Representations of the interactions among components are restricted to defined connections established between interface elements. Connections establish one of the following interactions:

- port connections (Sections 8.1 and 8.2): These are explicit relationships declared between **ports** or between **port groups** that enable the directional exchange of **data** and **events** among components.

- component access connections (Section 8.3): These are explicit declarations that enable multiple components access to a common **data** or **bus** component.

- subprogram calls (Section 8.4): These are explicit declarations within component implementations that enable synchronous call/return access to **subprograms**.

- parameter connections (Section 8.5): These are relationships among data elements associated with **subprogram calls**.

Interface elements are declared within the **features** section of a component type declaration. Paths of interaction (i.e., connections) between interface elements are declared explicitly within component implementations.

8.1 Ports

A **port** represents a communication interface for the directional exchange of **data**, **events**, or both (**event data**) between components. Ports are classified as

- **data port**: interfaces for typed state data transmission among components without queuing
  Data ports are represented by typed variables in source text. The structure of the variable/array is defined by the **data** type [**data classifier**] on the ports. Connections between data ports are either immediate or delayed.

- **event port**: interfaces for the communication of **events** raised by **subprograms**, **threads**, **processors**, or **devices** that may be queued
  Examples of **event port** use include: triggers for the dispatch of an aperiodic **thread**, initiators of mode switches, and alarm communications. Events such as alarms may be queued at the recipient, and the recipient may process the queue content. Event ports are represented by variables within source text that are associated with runtime service calls.

- **event data port**: interfaces for message transmission with queuing
  These interfaces enable the queuing of the **data** associated with an **event**. An example

of **event data port** use is modeling message communication with queuing of messages at the recipient. Message arrival may cause dispatch of the recipient and allow the recipient to process one or more messages. These ports are represented by **port** variables in source text that are associated with relevant runtime service calls.

8.1.1　Port Declarations

Ports are declared as **features** in component type declarations. Ports are directional. An **out** port represents a component's output and an **in** port represents a component's input. An **in out** port represents input and output to a component that maps to a single static variable. An **in out data** port represents both an incoming and an outgoing port such that the outgoing and incoming connections can be made to different components.

The graphical representations for data ports, event ports, and event data ports are summarized in Figure 8-1.
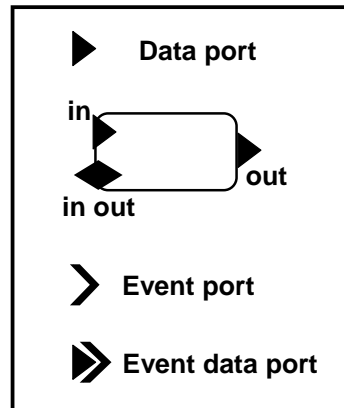
Figure 8-1: Port Graphical Representations

Table 8-1 has an example textual specification and corresponding graphical representation that includes port and port connection declarations. Within component type specifications, appropriate ports declarations are grouped together in the **features** section. Supporting **data** type definitions are included at the end of the table. Many of the other details of the specification are not shown. For example, declarations of **data** types used in data port declarations are not included, as in the declaration of the port c_data_out where the declaration of the **data** type processed_data is not shown.

In addition to user-defined ports, there are implicitly declared ports for threads.[18] For example, Error is an implicitly declared **out event data port** for all threads and may be declared as part of a connection involving a **thread**. In addition, there is an implicit Complete **out event port** that, if connected, raises an **event**, signaling the completion of a **thread**. Implicit ports can be used directly in connection declarations. They are not included in a **features** subclause.

---

[18]　The predeclared ports for a thread are Dispatch, Complete, and Error [SAE 06a].

8.1.2   Port Connections

Connection declarations between ports are also shown in Table 8-1. A connection declaration consists of

1.   optional identifier (name)
2.   colon (:)
3.   port connection descriptor (e.g., **data port**)
4.   source port
5.   connection symbol (e.g., the symbol -> for an immediate connection)
6.   destination port

The pattern for port connection textual declaration is shown in the box below:

name : [**descriptor**] [**source port**] [connection symbol] [**destination port**]

Graphically, **connections** are solid lines between the ports involved in the connection, sometimes with adorned with double cross hatching. See Section 8.1.5 (Immediate and Delayed Communications).

For example, in Table 8-1, the connection c_data_transfer is between the **out data port** c_data_out of the **thread** input (written as input.c_data_out) and the **in data port** c_data_in of the **thread** control_plus_output (written as control_plus_output.c_data_in). The **connections** declaration brake_in: **event port** brake -> input.brake_event; connects the **in event port** brake of **process implementation** control.speed_control to the **in event port** brake_event of the **thread** subcomponent input. A name for the **data port** connection between control_plus_output.c_cmd_out and throttle_cmd is not included in this example. The implicit **event data port** Error is used in the connection error_connection. It is connected to the **out event data port** Error_Signal but not declared explicitly as a feature in the originating **thread**.

Table 8-1: Sample Declarations of Data, Event, and Event Data Ports

```
process control
features
speed: in data port raw_speed;
brake: in event port;
set_speed: in event data port raw_set_speed;
throttle_cmd: out data port command_data;
Error_Signal: out event data port;
end control;

thread control_in
features
speed_in_data: in data port raw_speed;
brake_event: in event port;
```

Table 8-1: Sample Declarations of Data, Event, and Event Data Ports (cont.)

```
set_speed_edata: in event data port raw_set_speed;
c_data_out: out data port processed_data;
end control_in;


thread control_out
features
c_data_in: in data port processed_data;

c_cmd_out: out data port command_data;
end control_out;


process implementation control.speed_control
subcomponents
input: thread control_in.input_processing_01;
control_plus_output: thread control_out.output_processing_01;
connections

speed_in: data port speed -> input.speed_in_data;
brake_in: event port brake -> input.brake_event;
set_speed_in: event data port set_speed -> input.set_speed_edata;
c_data_transfer: data port input.c_data_out ->
                                    control_plus_output.c_data_in;

data port control_plus_output.c_cmd_out -> throttle_cmd;
error_connection: event data port input.Error -> Error_Signal;
end control.speed_control;

thread implementation control_in.input_processing_01
end control_in.input_processing_01;

thread implementation control_out.output_processing_01
end control_out.output_processing_01;

data raw_speed
end raw_speed;

data raw_set_speed
end raw_set_speed;

data command_data
end command_data;

data processed_data
end processed_data;
```
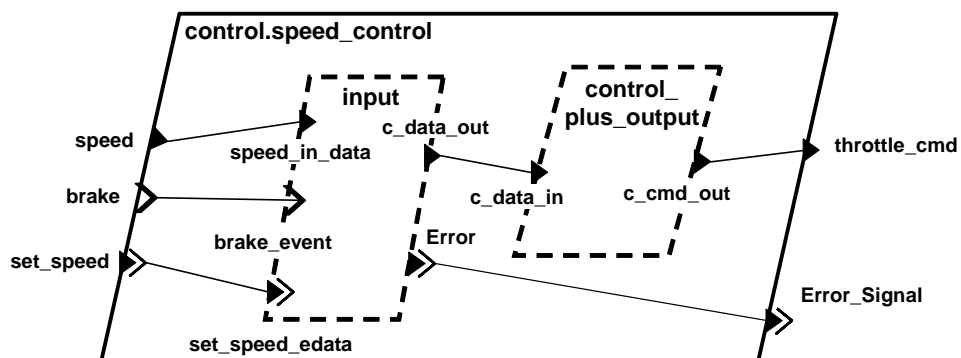
8.1.3 Connections in System Instance Models

A *connection instance* represents the actual flow of data and control between components of a **system** instance model. In case of a fully specified **system**, this flow is a transfer between two **thread** instances, a **thread** instance and a **processor** instance, or a **thread** instance and a **device** instance. The **data** flow may be in either direction. However, at least one **thread** must be included. In the AADL standard, connection instances in a fully specified **system** model are called *semantic connections*.

In the case of a partially specified **system**, the **system** instance model is expanded through the component hierarchy to the subcomponents for which no implementation detail is provided, regardless of their component category. In this case, connection instances may be between ports of **system** component instances or **process** component instances. According to the AADL standard, those connection instances are not semantic connections, but they are essential to certain analyses of partial **system** instance models.

Connection instances that are semantic connections are illustrated in Figure 8-2. In this figure, **data** is communicated between two threads in different processes. The **data** connection between the two threads is expressed by connection declarations that must follow the component hierarchy. In other words, there is a connection declaration from the original **thread** to its enclosing **process**, from that **process** to the second **process**, and from that **process** to the contained destination **thread**. Note that threads cannot arbitrarily communicate with other threads in the system. The enclosing **process** determines, through the ports in its type declaration and the connection declarations to those ports, which **data** from its **threads** should be passed on to **threads** in other processes.

In a **system** instance model, the sequence of **data** connection declarations from a **thread** to its enclosing **process**, to the second **process**, and to the **thread** contained in the second **process** results in a connection instance. If two **threads** are subcomponents within the same **process** or **thread group,** the connection instance is represented by a single connection declaration between those threads in the enclosing component **implementation**. While there may be a series of port-to-port connections involved in a **data** transfer (**system** instance connection) between two threads, **data** is transferred directly from the sending **thread** to the receiving **thread**. From an application source code perspective, the sending **thread** assigns a value to a variable/array and the receiving **thread** receives that value in a corresponding variable/array.
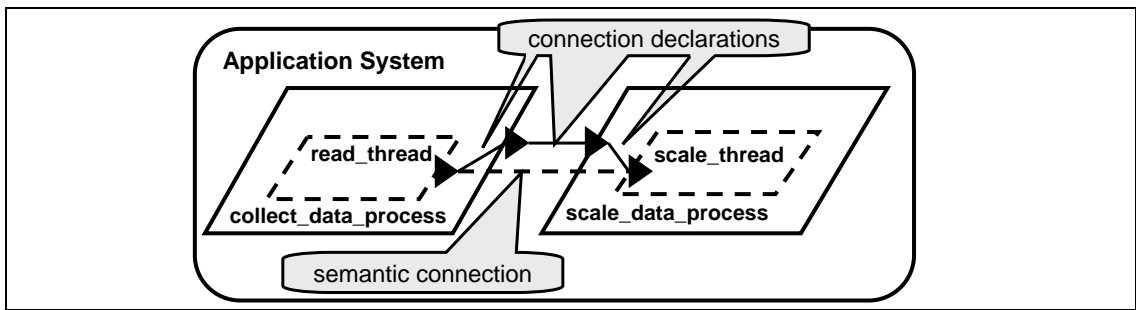
Figure 8-2:   A Semantic Connection between Thread Instances

Figure 8-3 illustrates a connection instance in a partial **system** instance model. In this model, the **data** collection **process** and the **data** scaling **process** have not been detailed out. The **data** connection between the two processes results in a connection instance in the **system** instance model. This connection instance is not considered a semantic connection according to the AADL standard, but the connection instance can be used in a fault propagation analysis or flow analysis of this partially specified **system**.
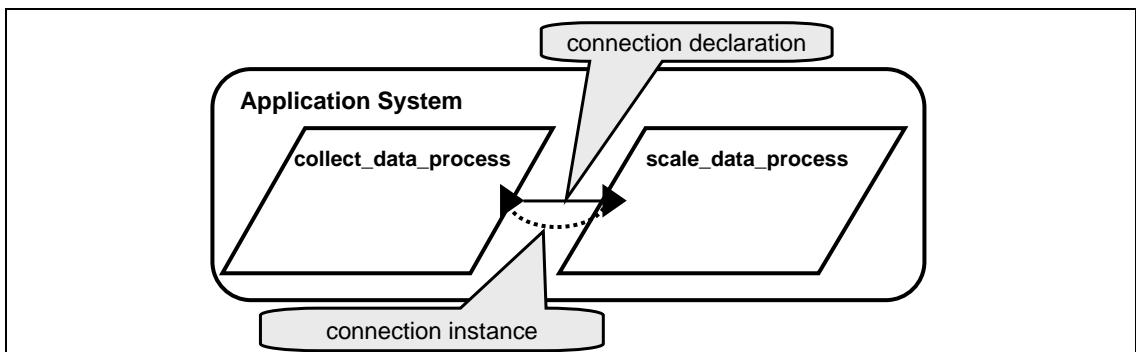
Figure 8-3:   A Connection Instance in a Partially Specified System Instance Model

8.1.4   Port Communication Timing

The timing of **system** instance **data** communication via ports depends upon the type of components involved (i.e., **thread, device,** or **processor**) and the nature of their **connections**. Communication timing is expressed in terms of execution completion, deadline, and dispatch times. For data port transfer out of threads, the **data** is ready for transfer at the completion of the **thread**, regardless of dispatch or scheduling characteristics. The timing of the delivery of the **data** to a receiving component is established by the nature of the data connection between them—immediate or delayed.

For **event** and **event data** ports, a source **thread** executes a Raise_Event call. This call results in the immediate transfer of control for an **event port** and the immediate transfer of both control and **data** for an **event data port**.

8.1.5 Immediate and Delayed Communications

The type of connection between **thread data ports** establishes specific timing semantics for **data** that is transferred between originating and terminating threads. Data port **connections** can be immediate or delayed. This section presents the basic timing semantics for these inter-thread **connections**. It does not address the potential impact of bus speeds, communication protocols, or partitions on these connections.

For immediate **connections**, **data** transmission is initiated when the source **thread** completes and enters the suspended state. The value delivered to the **in data port** of a receiving **thread** is the value produced by the sending **thread** at its completion. For an immediate connection to occur, the threads must share a common (simultaneous) dispatch. However, the receiving **thread's** execution is postponed until the sending **thread** has completed its execution. This aspect can be seen in Figure 8-4, where the immediate connection specifies that the **thread** control must execute after the **thread** read_data, within every 50 ms period. In addition, the value that is received by the **thread** control is the value output by the most recent execution of the **thread** read_data.
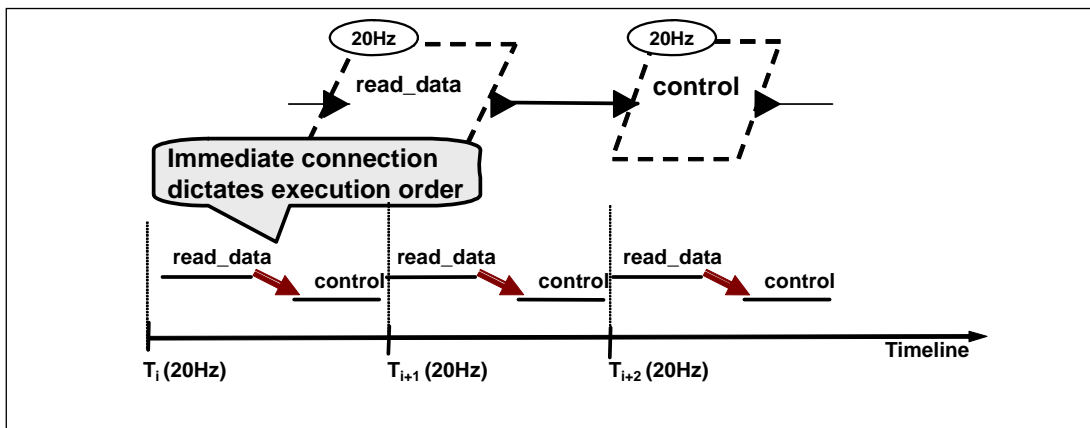
Figure 8-4:   An Immediate Connection

For the graphical timelines in Figure 8-4 through Figure 8-9, a horizontal bar above the timeline that is labeled with a **thread** name represents the execution time of that **thread**. The left edge represents the start and the right edge represents the termination of the **thread's** execution. A solid or segmented arrow between **thread** execution bars represents a **data** transfer between threads. A segmented arrow represents a delayed (e.g., Figure 8-5) or a repeat transfer (e.g., Figure 8-6).

For the two threads illustrated in Figure 8-4, a partial textual specification is shown in Table 8-2. The connection immediate_C1 is declared as immediate using the single-headed arrow symbol (**->**) between the **out data port** and **in data port**. Notice the Period **property** association (50 ms) within each of the **thread** type declarations.

Table 8-2: AADL Specification of an Immediate Connection

```
thread read_data
features
in_data: in data port;
out_data: out data port;
properties
Period => 50 ms;
end read_data;
--
thread basic_control
features
in_data: in data port;
out_data: out data port;
properties
Period => 50 ms;
end basic_control;
--
process implementation control_speed.impl
subcomponents
read_data: thread read_data;
control: thread basic_control;
connections
immediate_C1: data port read_data.out_data -> control.in_data;
end  control_speed.impl;
```

For a delayed port connection, the value from the sending **thread** is transmitted at its deadline and is available to the receiving **thread** at its next dispatch. For delayed port connections, the communicating threads do not need to share a common dispatch. In this case, the data available to a receiving **thread** is that value produced at the most recent deadline of the sending **thread**. If the deadline of the sending **thread** and the dispatch of the receiving **thread** occur simultaneously, the transmission occurs at that instant. The impact of a delayed connection can be seen in Figure 8-5, where the **thread** control receives the value produced by the **thread** read_data in the previous 50 ms frame. A shown in Figure 8-5, a delayed connection is symbolized graphically by double cross hatching on the connection arrow between the ports.

For the two threads illustrated in Figure 8-5, a partial textual specification is shown in Table 8-3. This specification has some differences from the one in Table 8-2: the connection delayed_C1 is declared as delayed using the double-headed arrow (**->>**) and the Period property association is declared in a **properties** subclause within the **process**. This association specifies that the value of 50 ms is the period of contained threads unless overridden within an individual thread's declaration.
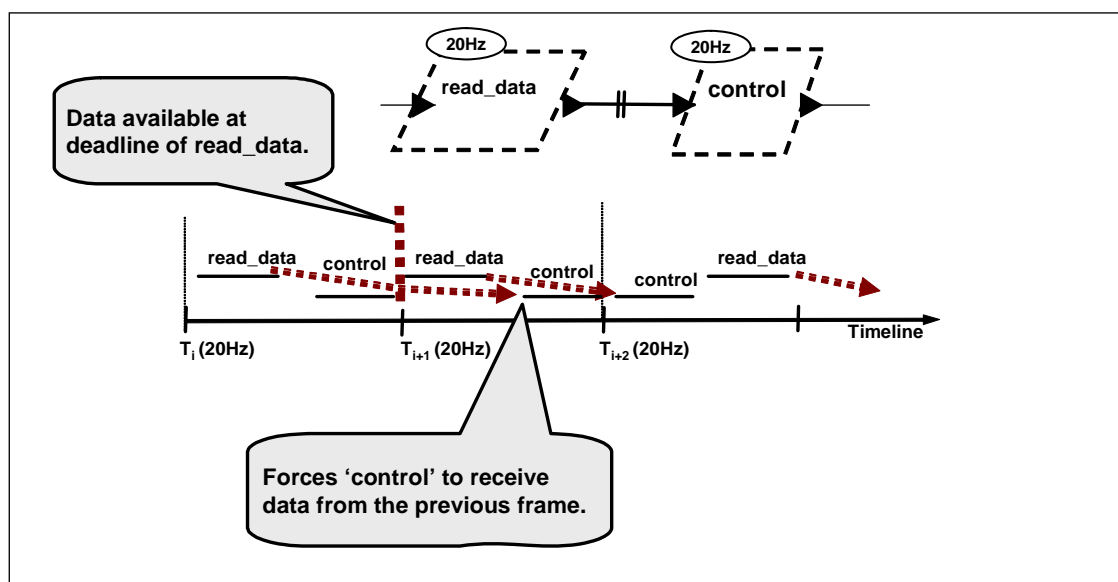
Figure 8-5:   A Delayed Connection

Table 8-3: AADL Specification of a Delayed Connection

```
Thread read_data
features
in_data: in data port;
out_data: out data port;
end read_data;
--
thread basic_control
features
in_data: in data port;
out_data: out data port;
end basic_control;
--
process implementation control_speed.impl
subcomponents
read_data: thread read_data;
control: thread basic_control;
connections
delayed_C1: data port read_data.out_data ->> control.in_data;
properties
Period => 50 ms;
end   control_speed.impl;
```

8.1.6   Oversampling and Under-Sampling

For communication between different frequency periodic threads with simultaneous dispatch, both delayed and immediate communications can be used to ensure a well-defined exchange.

Consider the example of two simultaneously dispatched threads `read_data` and `control` shown in Figure 8-6 and Figure 8-7. In the case of a delayed connection, the value from `read_data` is available at its deadline. It is received by the two executions of `control` whose dispatch coincides with or follows that deadline (e.g., `read_data` may have a

preperiod deadline). Thus, the two executions of control occurring within an execution frame of read_data receive the value produced in the preceding frame of read_data.

In contrast, consider the case of immediate connections as shown in Figure 8-7, the values available for two sequential executions of control are the same, the value produced within the 10 Hz execution frame of read_data. This result is accomplished by delaying the execution of the first control within the frame until the completion of read_data. Notice that this can only occur if both read_data and an execution of control can successfully complete (i.e., meet deadline) within the execution frame of control.
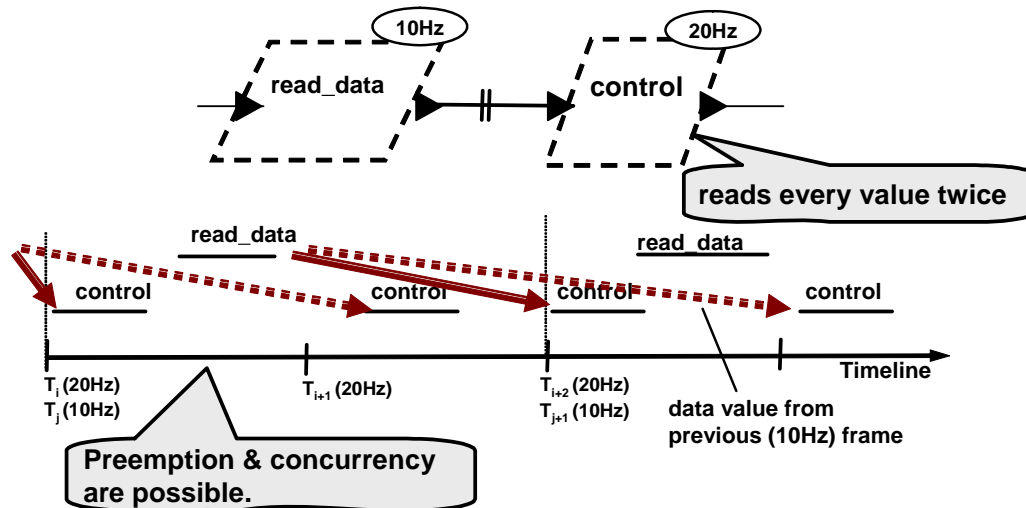
Figure 8-6:   Oversampling with Delayed Connections
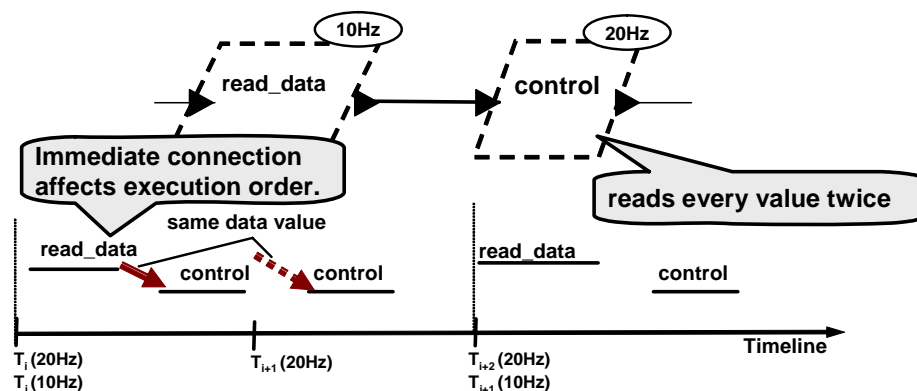
Figure 8-7:   Oversampling with Immediate Connections

Consider the situation where a periodic **thread** is sending to a simultaneously dispatched higher frequency **thread**. For a delayed connection, as shown in Figure 8-8, the data provided to an execution of control is the value produced by read_data that is available at the simultaneous dispatch of the threads. That value is produced at the most recent read_data deadline, which may coincide with the thread's dispatch. In the case of an immediate connection as shown in Figure 8-9, the value provided to the **thread**

control is the value produced by read_data at the end of its first execution after the simultaneous dispatch, and the execution of control is delayed until read_data has completed.
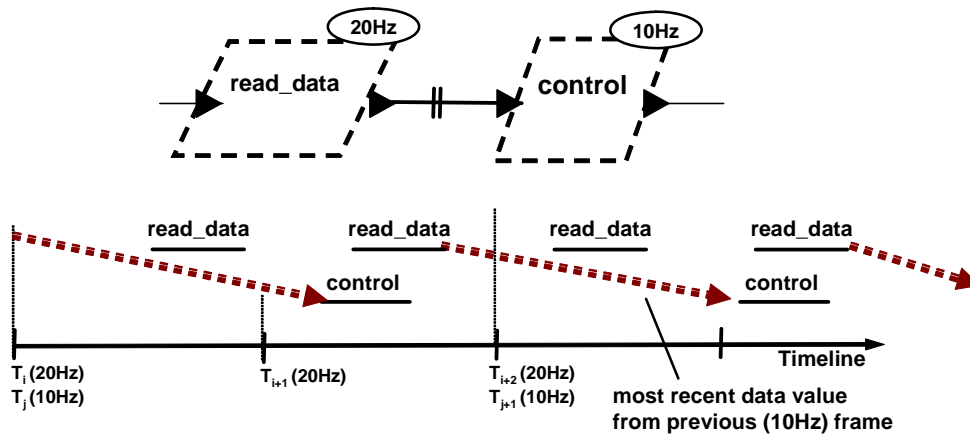
Figure 8-8:   *Under-Sampling with Delayed Connections*
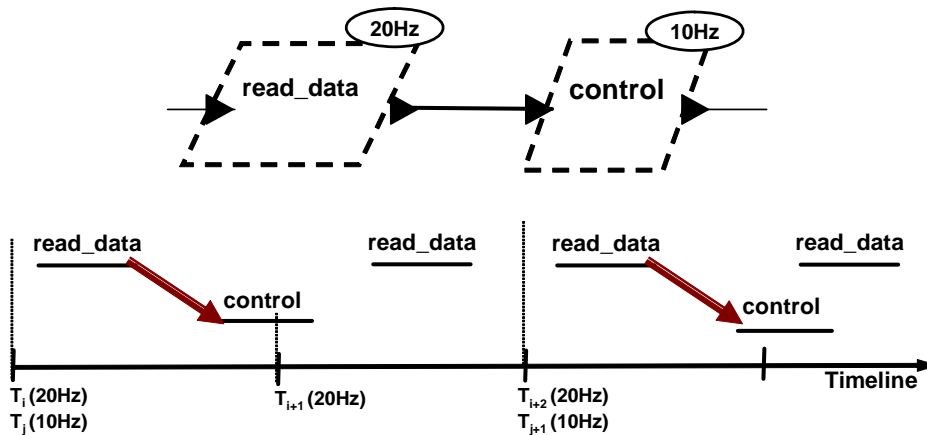
Figure 8-9:   *Under-Sampling with Immediate Connections*

8.1.7 Properties

A variety of predeclared port **properties** provide details on the interface represented by the port, including **properties** relating to the

- source text for the port
- whether a connection is required for the port
- port binding characteristics
- entry points associated with event and event data ports

For example, Source_Name is used to specify the name of the port variable in the source code. Required_Connection is used to indicate whether the component's **implementation** is aware of a port's having a connection (i.e., the connection may be

optional) or whether the component assumes the connection to always be in place.[19] Port-specific execution time, deadline, and source code entrypoints can be specified for each port to reflect that each may cause a different piece of code to be executed. Several properties allow the queue characteristics of event and event data ports to be specified.

In addition, predeclared port connection **properties** allow the declaration of specific connection protocols and **binding properties** relating to the connection. **Binding properties** allow the declaration of actual and allowed binding as well as the specification of restrictions on the co-location of software elements associated with the connection.

8.1.8   Port and Port Connection Constraints

There are restrictions on the topology of port connections. An **out data port** can be connected to (i.e., send data to) data ports of multiple components—a "fan-out" of data. An **in data port**, however, is restricted to a single incoming connection. In other words, because it does not support queuing, an **in data port** cannot have a "fan-in" from different sources; the outputs from those sources would overwrite one another. If queuing of data is desired, an **event data port** should be used. In contrast, event ports and event data ports support both data fan-out and fan-in. Fan-in is supported because these ports support queuing. Multiple inputs at an **event** or **event data port** enable the specification of the sequencing of disparate events as well as the queuing of events.

While it is permissible to omit the explicit declaration of the data type for a **data** or **event data port**, the explicit declaration allows checking of consistency of data type and size for the connections made between ports. Thus, the connection from the **out data port** of the **thread** read to the **in data port** of the **thread** scale in Figure 8-3 requires that the **data** type declaration for each of these ports and all of the intervening ports must be the same for a complete system specification. However, incomplete port specifications are permitted. For example, it is acceptable for one end of a connection not to have a **data** type declared while the other end does. Similarly, one end of a connection can have just a **data** component type while the other end has a **data implementation** with the same type.

8.2   Port Groups

The **port group** abstraction represents a collection of ports or other port groups. The content and structure of a **port group** are declared completely through a **port group** type declaration. There is no **implementation** declaration. Port groups are declared in the **features** section of component types and reference a **port group** type. They may be incompletely specified by not referring to a **port group** type or by referring to a **port group** type containing ports that themselves are not completely specified.

---

[19]   Source_Name and Required_Connection are in the predeclared property set
     AADL_Properties that is part of every AADL specification [SAE 06a].

Port groups can be used to

- reduce the number of connection declarations

- simplify graphical presentations

- allow a single reference to multiple related ports, connections, and entities in a specification

- group ports with common properties (e.g., all event ports)

- mix port types and directions

8.2.1   Port Groups and Port Group Type Declarations

A **port group** is defined in a type declaration that explicitly identifies the individual ports and port groups that it comprises. Example **port group** declarations and their declaration as **features** within a component type are shown in Table 8-4. As with other component type declarations, **properties** of the **port group** can be declared and a **port group** type can be extended and refined.

The declarations in the Table 8-4 are excerpts from a complete specification and include only relevant declarations and portions of declarations needed to show what is required in specifying a specific **port group**. In the tables, **port group** type declarations are shown in the left column and example references to the type and supporting declarations are shown in the right column.

Table 8-4: Sample Port Group with Mixed Port Types

port group type declaration	port group reference (with supporting declarations)
port group roll_set **features** roll_data: **in data port**; roll_cmd: **out data port** c_form; engage: **in event port**; errors: **port group** error_set; **end** roll_set; **data** c_form **end** c_form; **port group** error_set **features** sensor_error: **in data port**; range_error: **out event port**; **end** error_set	**process** control **features** roll_01: **port group** roll_set; **end** control;

A **port group** type can be declared as the inverse of another **port group** type. This relationship is indicated by the reserved words **inverse of** and the name of a **port group** type. The **features** of the inverted **port group** must be in the same order as in

the referenced **port group** but with the opposite directions. A **port group** type that is named in an **inverse of** statement cannot itself contain an **inverse of** statement. Thus, a chaining of inverses, such as *B inverse of A* and *C inverse of B*, is not permitted. An example of the use of the key word **inverse of** is shown in Table 8-5.

Table 8-5: A Port Group Type Declaration and its Inverse

```
port group GPS_socket
features
      Wakeup: in event port;
      Observation: out data port position;
end GPS_socket;

port group GPS_plug
features
      WakeupEvent: out event port;
      ObservationData: in data port position;
inverse of GPS_socket
end GPS_plug;
```

Figure 8-10 contains graphical icons for port groups and their connections. The graphical symbols of a **port group** represent the **features** declaration of the **port group** within a component type declaration. Port groups can bundle different port types and directions.

Port Group (as a feature of a thread)	
Port Group Connection (between two port groups that are each a feature of system)	
Port Group Bundle (mixed directions and ports)	Port group
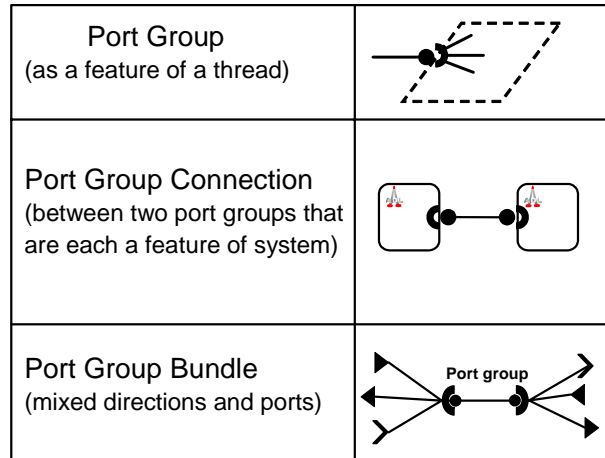
Figure 8-10:   Graphical Representations of Port Groups

8.2.2   Port Group Connections

Connections can be made between **port groups**, individual **ports**, and the individual **ports** within a **port group**. Within a component, elements of a **port group** in its component type can be individually connected to ports of subcomponents**.** However, elements of a **port group** of a subcomponent cannot be individually connected to other subcomponents. In other words, grouping and pulling apart elements of a **port**

group can occur when going up or down the component hierarchy, but not within the same level of the component hierarchy.

Figure 8-11 shows a graphical representation of a **port group** identified as mode_control_group and its inverse, with relevant excerpts from a corresponding AADL specification for a simple cruise control system. The connection declaration between the port groups is shown in Table 8-6 that includes excerpts from an AADL specification.
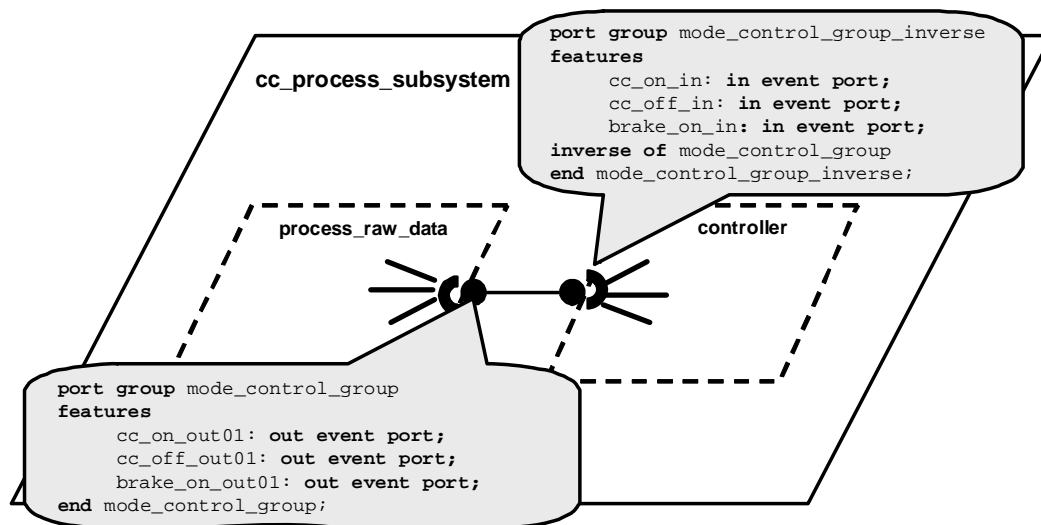
Figure 8-11:   Sample Port Group Connections

Table 8-6: Sample Port Group Connection Declarations

```
process implementation process_subsystem.cc_process_subsystem
…
subcomponents
process_raw_data: thread process_data.cc_process_raw_data;
controller: thread control.cc_control;
connections
d_to_c: port group  process_raw_data.mc_out -> controller.mc_in;
…
end process_subsystem.cc_process_subsystem;
…
thread process_data
features
mc_out: port group mode_control_group;
end process_data;
…
thread control
features
mc_in: port group mode_control_group_inverse;
end control;
```

Port groups can be effective in grouping related **data** and **connections**. For example, the individual outputs of multiple sensors (devices) within a sensor subsystem (grouped in a system) can be bundled together into a single **port group**. In that instance, all of the

sensor data is transferred through a single connection declaration from the sensor subgroup to a control processing system. The information provided by the ports within the **port group** is distributed through separate connections to individual control processing subsystems.

8.2.3 Aggregate Data Ports

Time consistency in data transmission can be achieved using an aggregate data **port group**. An aggregate data **port group** consists exclusively of data ports that have the same direction (i.e., all **out data ports**) with an Aggregate_Data_Port property value of *true*.[20] For this specialized **port group**, data transmission from multiple ports is time coordinated—that is, if data associated with the **port group** is produced by a set of simultaneously dispatched periodic threads, the recipients of that data receive a consistent set of values from the most recent dispatch or a consistent set of values from the previous dispatch of the threads.

8.2.4 Properties

Predeclared **port group** properties can be used to establish a **port group** as an aggregate data **port** and define **port group** memory binding characteristics. **Port group connections** can have properties that reflect the properties of the ports that compose the **port group**. For example, there is a Source_Text **property** that specifies the source files associated with the **port group** and an Allowed_Memory_Binding property that specifies the set of **memory** components to which **data** and **event data ports** within the **port group** can be bound.

8.3 Subcomponent Access

Data and **bus** subcomponents are made accessible throughout a **system** through explicit **features** declarations within type declarations of components. For **data** components, this capability supports modeling of shared access to a common data area or static data. For **bus** components, this **access** models the connectivity of execution platform components through buses whose access they share.

The **access** declarations are

- **provides**: indicates that a component provides access to a data or bus component contained within it
- **requires**: indicates that a component requires access to a data or bus component that is external to it

---

[20] Aggregate_Data_Port is a predeclared property for every AADL specification [SAE 06a].

8.3.1 Data Access Declarations

Examples of a **data** subcomponent **access** declaration are shown in Table 8-7. There is an optional identifier for the declaration.

Table 8-7: Data Access Declarations

```
process control
features
cc_set_point_data: requires data access data_sets.set_points;
error_log_data: provides data access logs.error_logs;
end control;

data data_sets
end data_sets;

data implementation data_sets.set_points
end data_sets.set_points;

data logs
end logs;

data implementation logs.error_logs
end logs.error_logs;
```

8.3.2 Data Access Connections

The connections (paths) for subcomponent **access** are declared in **connections** declarations within component implementations. The access connection specifies the path from the component providing access to the component requiring access (i.e., from **provides** to **requires**).

Table 8-8 presents an example of **data access connections** declarations. The lower portion of Table 8-8 is a graphical representation of these data access dependencies. The example shows some of the declarations for the **system implementation** basic_control.auto_cc that are relevant to the data access relationships for the system. The **thread** subcomponent cc_algorithm of the **process** cc_control requires access to the local **data** subcomponent comm_error_log (logs.error_logs). In addition, the **thread** subcomponent comm_errors requires access to the **data** subcomponent comm_error_log (logs.error_logs) of the **process** cc_error_monitor. This connection is a remote connection across address spaces, where the **process** cc_control provides access to its **data** subcomponent.

Notice the concurrent access to the **data** subcomponent comm_error_log (logs.error_logs) in the example. The predeclared **property** Concurrency_Control_Protocol can be used to coordinate this access (e.g., to ensure mutually exclusive access). Other predeclared **properties** for data subcomponent access identify whether the required or provided access is read_only, write_only, or

read_write. A Required_Access **property** association must be the same as the
Provided_Access **property** of the component that is accessed.[21]

Table 8-8: Shared Access across a System Hierarchy

```
system implementation basic_control.auto_cc
subcomponents
cc_control: process control.cc_control;
cc_error_monitor: process monitor.error_monitor;
connections
a_01: data access cc_control.error_log_data ->
cc_error_monitor.error_data_in;
end basic_control.auto_cc;
--
process control
features
error_log_data: provides data access logs.error_logs
                  {Provided_Access => access read_only;};
end control;

process implementation control.cc_control
subcomponents
comm_error_log: data logs.error_logs {Provided_Access =>
                                        read_write;};
cc_algorithm: thread algorithm.cc;
connections
data access comm_error_log -> error_log_data;
data access comm_error_log -> cc_algorithm.error_log_data;
end control.cc_control;

thread algorithm
features
error_log_data: requires data access logs.error_logs
                      {Required_Access => access read_write;};
end algorithm;

thread implementation algorithm.cc
end algorithm.cc;

data logs
end logs;

data implementation logs.error_logs
end logs.error_logs;

process monitor
features
error_data_in: requires data access logs.error_logs
                      {Required_Access => access read_only;};
end monitor;
```

---

[21] The predeclared properties Concurrency_Control_Protocol, Required_Access, and
Provided_Access are included in the property set AADL_Properties. This property set
declaration is part of every AADL specification [SAE 06a].

Table 8-8: Shared Access across a System Hierarchy (cont.)

```
process implementation monitor.error_monitor
subcomponents
comm_errors: thread m_algorithm.errors;
end monitor.error_monitor;




thread m_algorithm
features
c_error_data: requires data access logs.error_logs
                          {Required_Access => access read_only;};
end m_algorithm;


thread implementation m_algorithm.errors
end m_algorithm.errors;
```
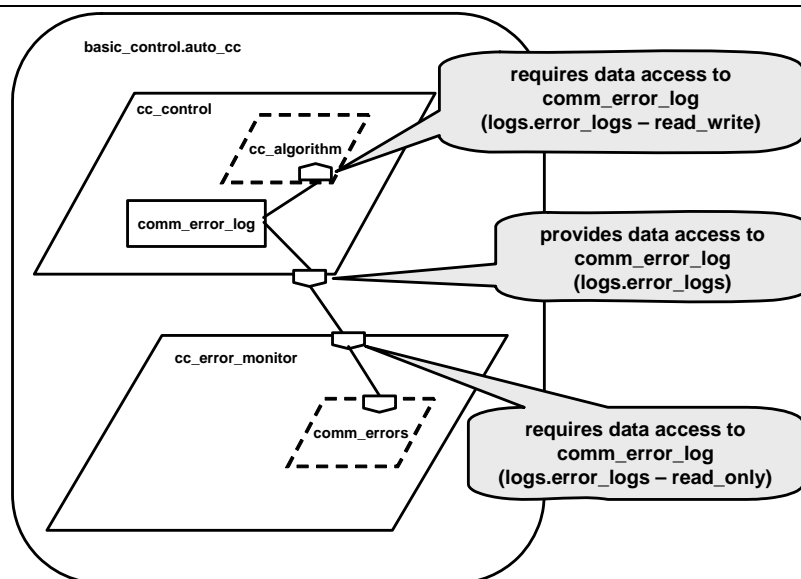

8.3.3 Bus Access and Bus Access Connections

In addition to access to data, access to buses is declared explicitly in AADL. Table 8-9 shows an example of **bus access** for a simplified cruise control system that consists of a cruise control unit (**system** component) and driver input, speed sensor, and throttle **devices**. The additional execution hardware for the system consists of a **processor** that executes the cruise control system application software and a **bus** connecting the hardware components. The figure in the lower portion of Table 8-9 is a graphical representation for required access **features** and **connections** to the **bus** declared in the text. It also shows the data connections for the system. Some of the details of the subcomponent declarations are not complete in the sample specifications.

Table 8-9: Basic Bus Access and Access Connection Declarations

```
system implementation cruise_control_system.impl
subcomponents
driver_input_unit: device driver_input_unit;
speed_sensor: device speed_sensor;
CCU: system CCU_system;
throttle_actuator: device  throttle_actuator;
M555: processor M555;
CANBus: bus CANBus.impl;
connections
-- data port connections not included
-- bus access connections
bus_access_01: bus access  CANBus ->  driver_input_unit.bus_access;
bus_access_02: bus access  CANBus ->  speed_sensor.bus_access;
bus_access_03: bus access  CANBus ->  throttle_actuator.bus_access;
bus_access_04: bus access  CANBus -> M555.bus_access;
end cruise_control_system.impl;
--

device driver_input_unit
features
set_speed: out data port;

bus_access: requires bus access CANBus.impl;
end driver_input_unit;
--
system cruise_control_system
end cruise_control_system;
--
bus CANBus
end CANBus;
--
bus implementation CANBus.impl
end CANBus.impl;
--
system CCU_system
end CCU_system;
--
device speed_sensor
features
bus_access: requires bus access CANBus.impl;
end speed_sensor;
--
device throttle_actuator
features
bus_access: requires bus access CANBus.impl;
end throttle_actuator;
--
processor M555
features
bus_access: requires bus access CANBus.impl;
end M555;
```

Table 8-9: Basic Bus Access and Access Connection Declarations (cont.)
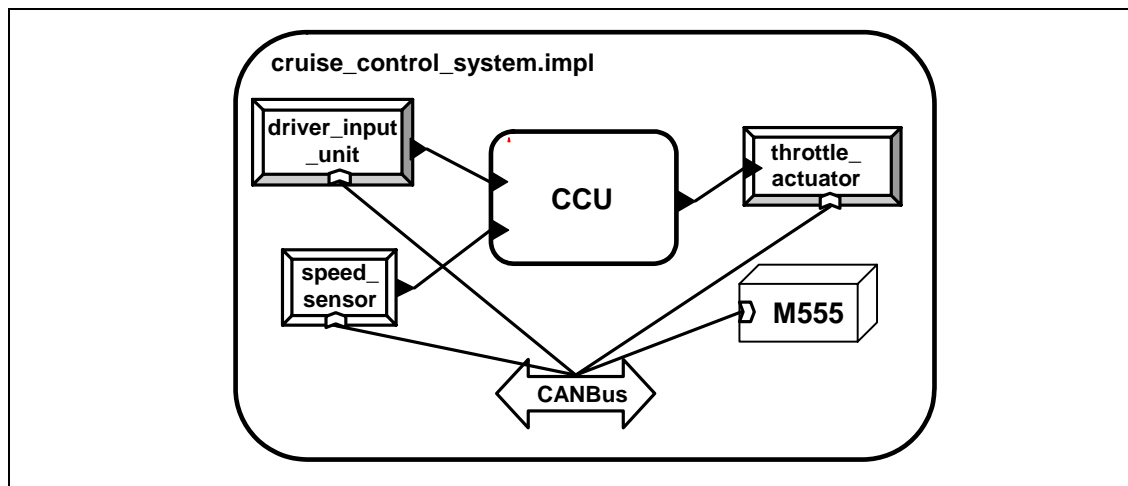
Table 8-10 illustrates how to model two subsystems with hardware components and **bus connections**. Some of the specifications are not complete (e.g., type rather than **implementation** classifiers are used in defining some of the components and subcomponents). In the illustration, one subsystem is connected to the other by a **bus** provided by the second subsystem. Specifically, the application system requires **bus access** to the network system's 1553 bus. The **bus access**, **requires**, **provides**, and **connections** are shown both graphically (lower portion of Table 8-10) and as AADL text declarations.

Table 8-10:   Example Bus Access Connection Declarations

```
system containing_system
end containing_system;
--
system implementation containing_system.impl
subcomponents
network: system network;
application: system application;
connections
bus access network.network_bus -> application.network_bus;
end containing_system.impl;
--
system network
features
network_bus: provides bus access B_1553;
end network;
--
system implementation network.impl
subcomponents
B_1553: bus B_1553;
connections
C01: bus access B_1553 -> network_bus;
end  network.impl;
--
```

Table 8-10: Example Bus Access Connection Declarations (cont.)

```
system application
features
network_bus: requires bus access B_1553;
end application;
--
system implementation application.impl
subcomponents
PC_processor: processor PC;
connections
bus access  network_bus -> PC_processor.network_bus;
end application.impl;
--
processor PC
features
network_bus: requires bus access B_1553;
end PC;
--
bus B_1553
end b_1553;
```
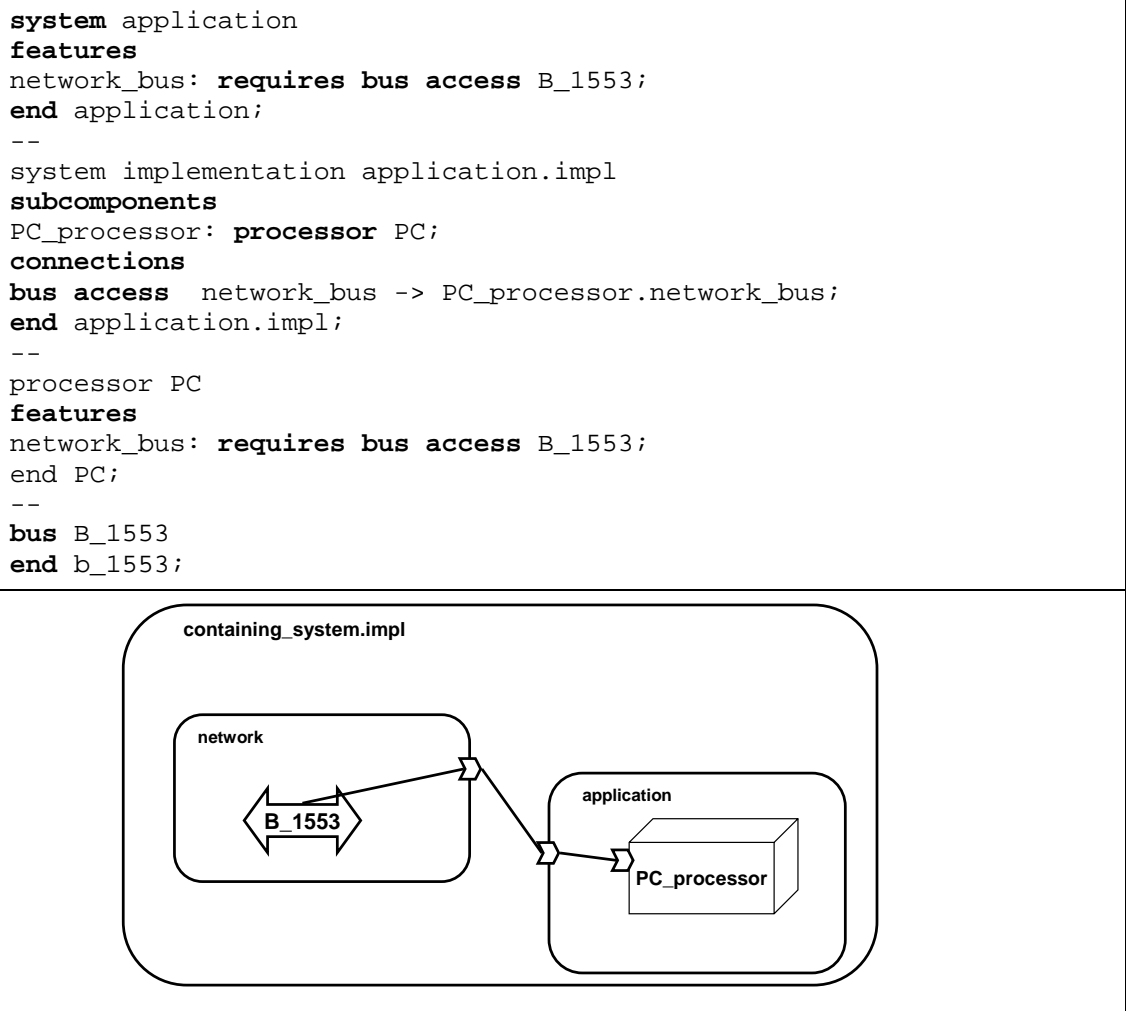

8.4   Subprogram Calls

Subprogram calls are declared through **calls** declarations within a **thread** or **subprogram implementation**. The **subprogram** that is called must be declared through a **subprogram** type declaration and possibly a **subprogram implementation** declaration, as discussed in the Section 5.5.1 (Subprogram Declarations).

In the current version of the AADL standard, subprograms are not declared as instances through a **subprogram** subcomponent declaration. The need for such instances is inferred from the **calls** and can take into account sharing of **subprogram** libraries across processes. The specific **subprogram** called is declared through a **property** association of the predeclared **property** Actual_Subprogram_Call. The example in Table 8-12 illustrates this principle.

8.4.1 Call Sequences

There may be a sequence of **calls** declared within a component **implementation**. An example is shown in the partial specification of Table 8-11 where the **calls** sequence two_calls involves a call to the **subprogram** implementations acquire.temp and then adjust.level. The associated **subprogram** declarations are also shown. The **calls** sequence is determined by the **subprogram calls** declaration order. In other words, the **calls** order is linear. If more complex call orderings are desired, an **annex** notation could provide specification of other orderings, such as a "branch" or "iteration." Alternatively, one can specify different **calls** sequences that are active under different **modes**. For more details on the use of **modes**, see Section 9 (Modes).

Notice that **subprograms** may call other **subprograms**. This circumstance is shown in Table 8-11 where the **subprogram implementation** adjust.level **calls** the **subprogram** find.temp_values.

Graphically, **subprogram calls** are represented by **subprogram** symbols, arranged left to right within a **thread implementation** or **subprogram** symbol. A call sequence arrow may be included as shown in the figure in the lower potion of Table 8-11.

Table 8-11:   Example Subprogram Calls

```
thread implementation control.thermal_control
--
calls
two_calls:{
                get_temp: subprogram acquire.temp;
                adjust_level: subprogram adjust.level;
        };
--
end control.thermal_control;

subprogram acquire
end acquire;

subprogram implementation acquire.temp
end acquire.temp;

subprogram adjust
end adjust;

subprogram implementation adjust.level
calls
      {
      find_scale_values: subprogram find.temp_values;
      };
end adjust.level;

subprogram find
end find;

subprogram implementation find.temp_values
end find.temp_values;
```

Table 8-11:   Example Subprogram Calls (cont.)
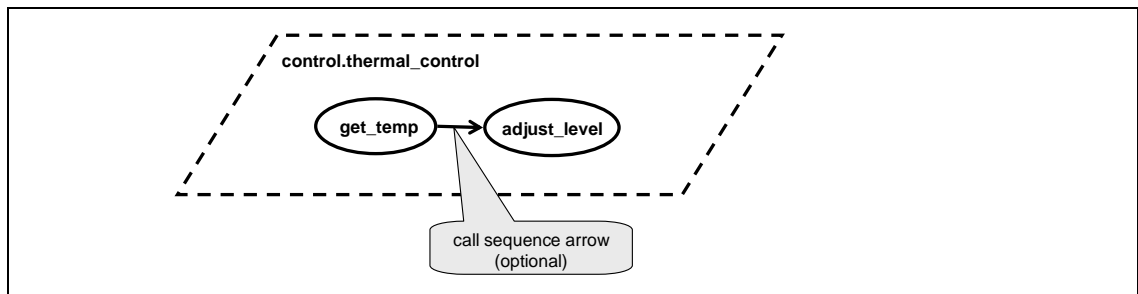
8.4.2 Remote Calls

Remote client-server interactions can be modeled using **server subprogram calls** as shown in the partial specification in Table 8-12. The **property** association `Actual_Subprogram_Call` declares that the **subprogram call** `call_server` within the **thread** `calling_thread`, which is a subcomponent of the **process** `client_process`, is being made to the **subprogram** contained within the **server process** (`server_process`). This is an example of a contained **property** association that is discussed in more detail in Section 11.2.2 (Contained Property Associations).

Table 8-12:   Client-Server Subprogram Example

```
system implementation client_server_sys.impl
subcomponents
client_process: process client_process.impl;
server_process: process server_process.impl;
properties
Actual_Subprogram_Call => reference server_process.
                                    server_thread.service
                             applies to  client_process.
                                         calling_thread.call_server;
end client_server_sys.impl;
--
process client_process
end client_process;
--
process implementation client_process.impl
subcomponents
calling_thread: thread calling.impl;
end client_process.impl;
--
thread calling
end calling;
--
thread implementation calling.impl
calls   {
            call_server: subprogram service_it ;
        };
end calling.impl;
----
```

Table 8-12:   Client-Server Subprogram Example (cont.)

```
process server_process
features
service: server subprogram service_it;
end server_process;
--
process implementation server_process.impl
subcomponents
server_thread: thread server_thread.impl;
end server_process.impl;
--
thread server_thread
features
service: server subprogram service_it;
end server_thread;
--
thread implementation server_thread.impl
end server_thread.impl;
--
subprogram service_it
end service_it;
```

client_server_system.impl

client_process

calling_thread

call_server

server subprogram call binding:
Actual_Subprogram_Call =>
  reference server_process.server_thread.service
  applies to client_process.calling_thread.call_server;

server_process

service   **server_thread**
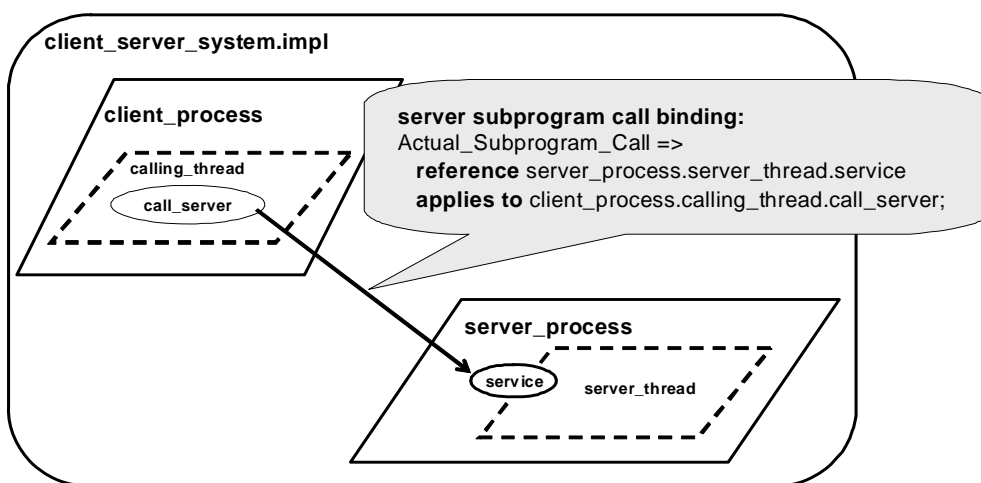
8.4.3   Properties

Subprogram calls properties identify the allowed and actual server subprograms involved in a remote **server subprogram** call. In addition, these **properties** can be used to specify the allowed and actual binding of the calls to physical elements that support a remote **server subprogram** call. If no values are assigned to these **properties**, the **subprogram** call is a local call to a **server subprogram**.[22]

---

[22]   In the AADL standard, the **subprogram calls** of all **threads** must either be local **calls** or be bound to a **server subprogram** whose **thread** is part of the same **mode**, in a completely instantiable **system** [SAE 06a].

8.5   Data Exchange and Sharing in Subprograms

A **subprogram** can receive and provide data through a variety of mechanisms including

- parameter (passing by value)
- access (passing by reference)
- global/static (shared) data

These diverse and often implicit aspects of data that are followed in programming languages can be modeled and explicitly documented in an AADL representation through parameters, access features, and their associated connections.

8.5.1   Data Exchange by Value: Parameters and Connections

A **parameter** represents call and return data values passed into and out of a **subprogram**. These exchanges by value are declared as typed **data features** in the type declaration of a **subprogram**, similar to **data port** declarations. **Parameter connections** are used to describe the flow of **data** into and out of a **subprograms** and the data flow through a sequence of **subprogram calls** within a **thread**. These **connections** can be useful in a comprehensive flow analysis when used in conjunction with **flows** declarations. For more detail on the use of parameters in flow analysis, see Section 10 (Flows).

Table 8-13 presents textual and graphical representations of the parameters and the **parameter connections** associated with a **calls** sequence within a **thread**. In a graphical representation

- **parameters** are represented as solid arrows (▶), like data ports
- **parameter connections** are shown as solid lines (▬) between parameters or between a **parameter** and a **port** (on a containing **thread** of the **subprogram** call)
- **subprogram calls** are represented by ovals (⬭) labeled with the call (e.g., scale) and called **subprogram** type
- **calls** sequence is indicated by an arrow with an open arrow head (→) (Alternatively, a **calls** sequence can be specified by the ordering of the **calls** from the left to the right.)

Notice that the **in event data port** in_data of the **thread** scale_data is connected to the **parameter** in_parameter of the **subprogram** scale. Parameters can be connected to **in data port, out data port**, and **event data port**.

Table 8-13:   Example Parameter Connections

```
thread scale_data
features
in_data: in event data port;
out_data: out data port;
end scale_data;
--
thread implementation scale_data.impl
calls   {
scale: subprogram scale;
edit: subprogram edit_range;
update: subprogram update_set;
        };
connections
parameter in_data -> scale.in_parameter;
parameter scale.interim_value -> edit.interim_value;
parameter edit.out_parameter -> update.io_parameter;
parameter update.io_parameter -> out_data;
end scale_data.impl;
--
subprogram scale
features
in_parameter: in parameter;
interim_value: out parameter;
end scale;
--
subprogram edit_range
features
interim_value: in parameter;
out_parameter: out parameter;
end edit_range;
--
subprogram update_set
features
io_parameter: in out parameter;
end update_set
```
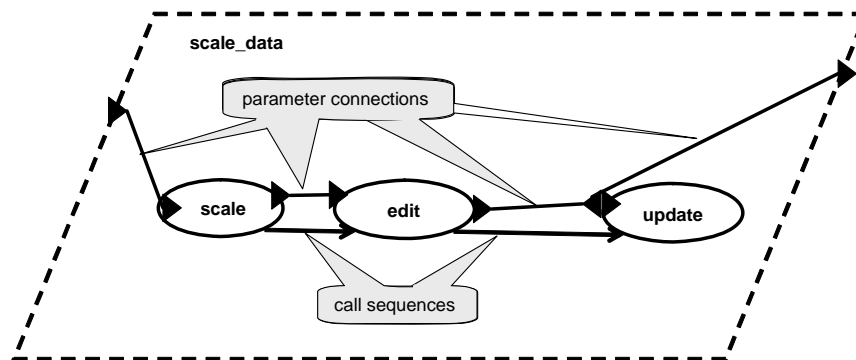

8.5.2 Data Passing by Reference and Global Data

The flow of **data** into and out of a **subprogram** can involve references to data (e.g., pointer values) or access to common data values (i.e., global or static data), rather than

explicit data passing. These **data reference** mechanisms are described through data **requires/provides data access** declarations in an AADL model.

For example, consider the annotated pseudocode and corresponding AADL textual representation in Table 8-14. In the pseudocode, examples of **subprogram calls** with **data reference** and the use of global data are shown. In the *Passing by reference* section of pseudocode, the function scale modifies data (referenced with the pointer p1) using the scale factor v1. In the second **implementation** of scale (the *Global variable* section of Table 8-14), a **parameter** data value (the scale factor) is passed and a common **data** element raw_data is scaled.

Within AADL, both of these options are represented with v1 as a **parameter**, whereas the pointer p1 and the common data raw_data are represented as a **data access** feature of the **subprogram** scale. The **thread** processing has a call to the **subprogram** scale. A corresponding AADL representation for the *Global variable* pseudocode explicitly shows the **thread** receiving the data value for v1 through the **in data port** scalar and using that value in the **subprogram** call, as indicated by the **parameter** connection VC1 in the **thread**. In contrast, the pointer reference to the data to be scaled is represented as a **data access** in the **subprogram** type declaration for scale. The explicit reference to raw_data in the **subprogram** scale is the **requires** statement in the **thread** type declaration. The AADL specification allows an **implementation** using either option shown in pseudocode.

Table 8-14:   Examples of Passing by Reference and Global Data

Pseudocode	AADL Representation

Passing by reference:
……
scale (v1, p1)
v1 is a real that is
the scale factor.
p1 is a pointer to a
data set 'raw_data'
that is to be scaled.
…
processing that calls
the subprogram:
…
call scale (v1, p1);
….
	```subprogram scale```
	```features```
	```v1: in parameter real;```
	```p1: requires data access raw_data;```
	```end scale;```
	```--```
	```data raw_data```
	```end raw_data;```
	```--```
	```data real```
	```end real;```
	```--```
	```thread processing```
	```features```
	```scalar: in data port real;```
	```p1: requires data access raw_data;```
	```end processing;```
	```--```

Table 8-14:   Examples of Passing by Reference and Global Data (cont.)

Global variable:	`thread implementation` processing.impl

…
 	
variable and processing definitions:
 	
….
 	
real: raw_data;
 	
…
 	
scale(v1)
    {
      x :=
raw_data;
    }
 	
…
 	
processing that calls the subprogram:
 	
…
 	
call scale(v1);
 	
….
	`thread implementation` processing.impl
	`calls` {
	        scale_it: `subprogram` scale;
	        };
	`connections`
	VC1: `parameter` scalar -> scale_it.v1;
	PC1: `data access` p1 -> scale_it.p1;
	`end` processing.impl;
	--
	`process` data_management
	`features`
	scalar: `in data port` real;
	`end` data_management;
	--
	`process implementation` data_management.impl
	`subcomponents`
	r_data: `data` raw_data;
	data_processing: `thread` processing.impl;
	`connections`
	C1: `data port` scalar -> data_processing.scalar;
	C2: `data access` r_data -> data_processing.p1;
	`end` data_management.impl;

8.5.3   Method Calls in AADL

`Calls` to object methods can be represented in AADL as `calls` to `subprogram features` of a `data` component. Consider the pseudocode in Table 8-15 where the method errorTotal of the class ErrorLog returns an integer value that is the total number of errors currently in the log. The corresponding AADL representation involves the declaration of an enclosing `process implementation` that establishes instances of the `thread` monitor and the `data` component ErrorData, as well as the required `data access` of the `thread` monitor to ErrorData. The `implementation` of the `thread` monitor involves the call sequence to subprograms errorTotal and reset. The integer type return value for errorTotal is represented as the `out parameter` total. The `data access connections` are shown graphically in the figure of Table 8-15 and indicate the `subprogram` and `thread` access to ErrorData.

Table 8-15:   Methods Calls on an Object

Object-Oriented Pseudocode	AADL Representation
| ```
class ErrorLog {
      int errorTotal () {
…
  }
    void reset() {
….
  }
….. 
``` | `process implementation`<br>maintenance.control<br>`subcomponents`<br>monitor: `thread` monitor.errors;<br>ErrorData: `data` ErrorLog;<br>`connections`<br>C1: `data access` ErrorData -><br>monitor.log_access;<br>`end`  maintenance.control; |

Table 8-15:   Methods Calls on an Object (cont.)

<table>
<tr>
<td>

```
public static void main()
{
…
ErrorLog stabilizer = new
ErrorLog();
int errors;
errors =
stabilizer.errorTotal();
stabilizer.reset();
...
 }
```

</td>
<td>

```
--
thread monitor
features
log_access: requires data access
ErrorLog;
end monitor;
--
thread implementation monitor.errors
calls {
errors: subprogram ErrorLog.errorTotal;
reset_it: subprogram ErrorLog.reset;
        };
Connections
  Data access log_access  ->
reset_it.this;
  Data access log_access -> errors.this;
end monitor.errors;
--
data ErrorLog
features
errorTotal: subprogram errorTotal;
reset: subprogram reset;
end ErrorLog;
--
subprogram errorTotal
features
  this: requires data access ErrorLog;
  total: out parameter
                  BaseTypes::integer;
end errorTotal;
--
subprogram reset
features
  this: requires data access ErrorLog;
end reset;
```

</td>
</tr>
<tr>
<td colspan="2">

</td>
</tr>
</table>

9 Modes

A **modes** abstraction is an explicitly defined configuration of contained components, **connections**, and **property value** associations. **Modes** represent alternative operational states of a system or component. For example, **modes** for a cruise control system may be {initialize, disengaged, engaged}, where each of these modes may involve different sets of processes, executing threads, or active connections (e.g., in the initialization mode there are no connections to sensors).

Modes may specify different **calls** sequences to be used in a **thread** or **subprogram**. **Modes** also may represent different logical states of any component, such as a **thread** or **subprogram**, for which different **property** values apply. For example, under different **modes** a **thread** may have different execution times to represent an algorithm that can execute with different levels of precision. **Modes** may also represent different hardware configurations such as processors that are active at any one time.

9.1 Modal Specifications

Modes are represented as states within a state machine abstraction. Each distinct configuration of a component is identified as one **mode** (state) within the modal state machine abstraction for the component. The configuration that defines each **mode** and the events that cause the transitions in the behavior of the component must be specified. Each modal state machine must have at least two modes, one of which must be declared as the **initial** mode for the component.

Modes can be used to represent alternative system configurations in a variety of ways. They can establish

- alternative configurations of active components and connections and the transitions among these configurations
- variable call sequences within a thread
- mode-specific properties for software or hardware components

9.1.1 Modal Configurations of Subcomponents and Connections

Table 9-1 presents both textual and graphical representations of **modes** transition specifications for a simplified controller **thread** within a cruise control system. In this example, mode transitions are triggered by external events. Only the relevant ports are shown in the type declaration for the **thread** control. Neither type nor **implementation** declarations are complete. The graphic shows the mode transition view for the **thread**.

There are two **modes**, idle and controlling, and three **event ports** in this example. The idle mode is the **initial** mode. The **event** brought into the **thread** by **event port** cc_engage results in a mode transition to the controlling mode (the **thread** configuration that provides the functionality to maintain a set speed). The **event** carried through the **event port** cc_resume_e1 also results in a switch to the controlling mode using the previous value of the speed setting. **Event port** cc_brake results in an exiting of the controlling mode to the idle mode.

Table 9-1: Sample Graphical and Textual Specifications for Modes

```
thread control
features
cc_engage : in event port;
cc_resume_e1 : in event port;
cc_brake: in event port;
end control;

thread implementation control.cc_control
modes
idle : initial mode;
controlling : mode;
idle -[ cc_engage, cc_resume_e1 ]-> controlling;
controlling -[ cc_brake]-> idle;
end control.cc_control;
```
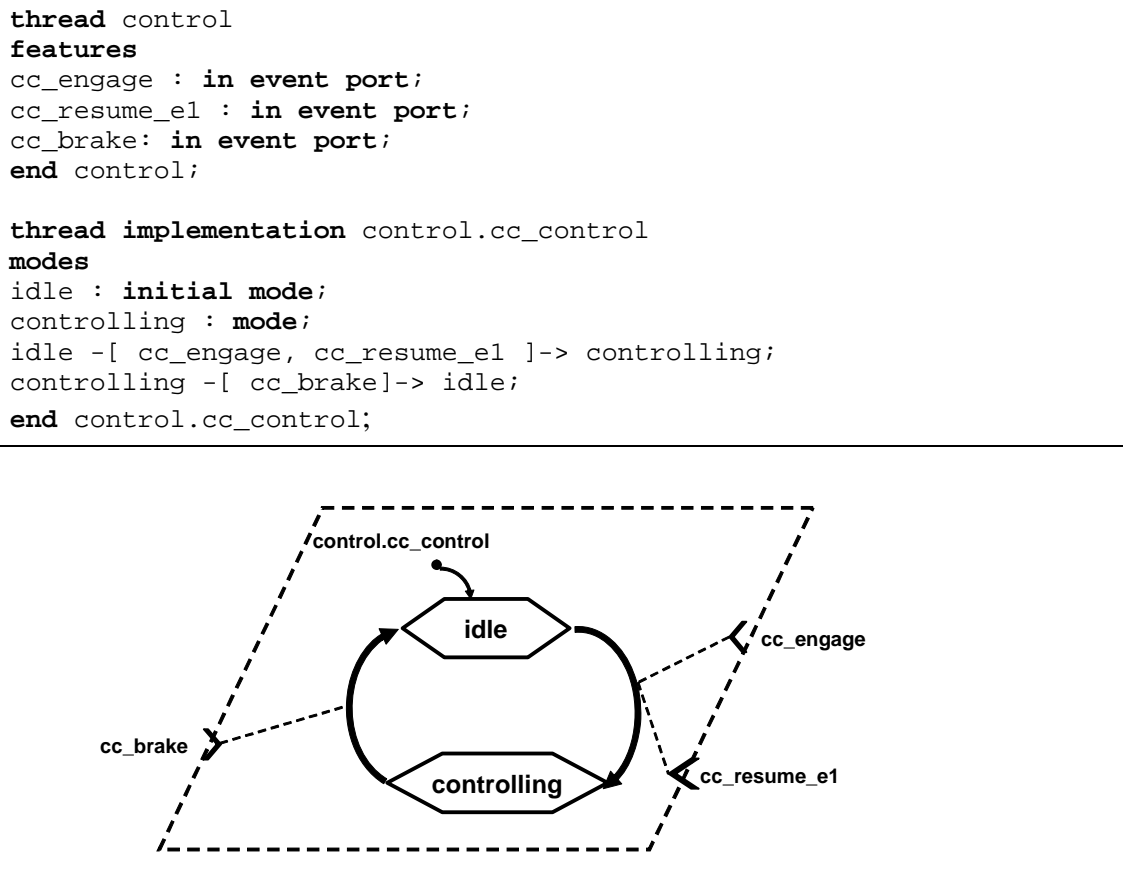

The example in Table 9-2 shows a multimode **process** where internal events result in mode changes of a **process**. In the textual specification for the **process** control_algorithms.impl, the **modes** section defines the two operational **modes** of ground and flight and the transitions between them. The transitions are triggered by **out event ports** from the **thread** controller that is a subcomponent of the **process implemenation** control_algorithms.impl. The specification for the **process implementation** includes **in modes** clauses that define the subcomponents and connections active in each **mode**.

In the upper right portion of the figure in Table 9-2, a graphic shows the **modes** and their transitions that are triggered by the **events** from the controller **thread**. In that figure, the flight **mode** configuration is shown in black and the ground **mode** is shown in gray. This distinction illustrates that the ground_algorithms **thread** and its **connections** are not part of the flight **mode**.

Table 9-2: Modes Example

```
process control_algorithms
features
status_data: in data port;
aircraft_data: in data port;
command: out data port;
end  control_algorithms;
--
process implementation control_algorithms.impl
subcomponents
controller: thread controller;
ground_algorithms: thread ground_algorithms in modes (ground);
flight_algorithms: thread flight_algorithms in modes (flight);
connections
C1: data port aircraft_data -> ground_algorithms.aircraft_data in
modes (ground);
C2: data port aircraft_data -> flight_algorithms.aircraft_data in
modes (flight);
C3: data port ground_algorithms.command_data -> command in modes
(ground);
C4: data port flight_algorithms.command_data -> command in modes
(flight);
modes
ground: initial mode;
flight: mode;
ground -[controller.switch_to_flight]-> flight;
flight -[controller.switch_to_ground]-> ground;
end control_algorithms.impl;
--
thread controller
features
status_data: in data port;
switch_to_ground: out event port;
switch_to_flight: out event port;
end controller;
--
thread ground_algorithms
features
aircraft_data: in data port;
command_data: out data port;
end ground_algorithms;
--
thread flight_algorithms
features
aircraft_data: in data port;
command_data: out data port;
end flight_algorithms;
```

Table 9-2: Modes Example (cont.)
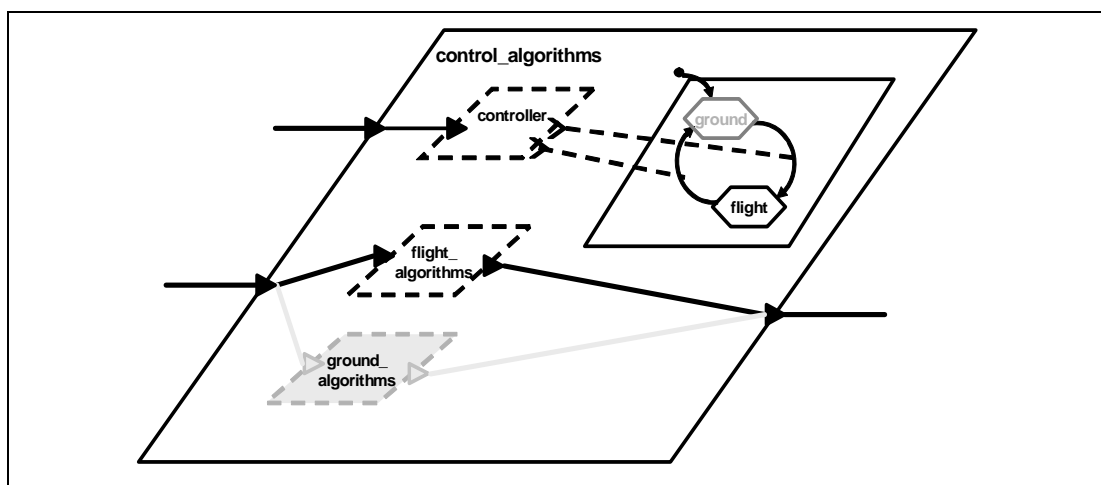
9.1.2 Modal Configurations of Call Sequences

Alternative **calls** sequences can be specified using **modes**. The example in Table 9-3 shows a monitor **thread** that checks software and hardware and reports anomalies. The **thread** employs a sequence of **calls** to subprograms when the **thread** is in the nominal **mode**. When an error is detected, an error_condition is signaled through the **event port** error_event. This signal results in a mode switch and changes the **subprogram calls** sequence of the **thread**.

Table 9-3: Mode-Dependent Call Sequences

```
thread monitor
features
error_event: in event port;
repaired: in event port;
end monitor;
--
thread implementation monitor.impl
calls
        nominal_sequence: {
                call_cksw: subprogram check_sw;
                call_ckhw: subprogram check_hw;
                call_report: subprogram report;
                } in modes (nominal);
        error_sequence: {
                call_alarm: subprogram alarm;
                call_diag: subprogram diagnose;
                callreport: subprogram report;
                } in modes (error_condition);
modes
nominal: initial mode;
error_condition: mode;
nominal -[error_event]-> error_condition;
error_condition -[repaired]-> nominal;
end monitor.impl;
```

9.1.3 Mode-Specific Properties

Property values assignments can be mode-dependent. These mode-specific **property** associations can be used to define alternative characteristics and behavior for components. For example, consider the partial specification in Table 9-4 that has a modified version of the **process implementation** for control_algorithms.impl shown in Table 9-2. In this example, the controller **thread** has a different execution time for the ground **mode** than for the flight **mode**.

Table 9-4: Mode-Specific Component Property Associations

```
process implementation control_algorithms.impl
subcomponents
controller: thread controller {Compute_Execution_Time => 2 ms..5ms
in modes (ground);
Compute_Execution_Time => 3 ms..7ms in modes (flight);};
ground_algorithms: thread ground_algorithms in modes (ground);
flight_algorithms: thread flight_algorithms in modes (flight);
--
end control_algorithms.impl;
```

10 Flows

AADL **flows** specification capabilities enable the detailed description and analysis of an abstract information path through a system. A complete **path** for an abstract information flow—an end-to-end flow implementation—begins at a **source** component and terminates at a **sink** component. The specification of an end-to-end flow involves the declaration of the elements of the **flow** (sources, sinks, and paths) and explicit **implementation** declarations that describe the details of a complete **path** through the system.

A **source** component of a **flow** is characterized by the feature (e.g., port, port group, or parameter) through which the flow emerges from the component. Similarly, a **sink** component of a **flow** is characterized by the feature through which the flow enters the component and terminates. Details of a **flow path** are described by identifying the entry and exit features of each intermediary component and subcomponent involved in the flow.

10.1 Flow Declarations

Flows are directional. To specify a complete flow, declarations in component types and implementations are required. For a component type, **flows** declarations designate a

- **source**: a feature of a component
- **sink**: a feature of a component
- **flow path**: a flow through a component from one feature to another

Table 10-1 shows a partial specification for a simplified cruise control system with **flow source**, **flow sink**, and **flow path** declarations within component type declarations. Notice that the **flow path** brake_flow through the **system** component cruise_control has an **in event data port** as its origin and an **out data port** as its termination feature. The lower portion of the table includes a graphical representation of the declarations.

Table 10-1:   Flow Declarations within a Component Type Declaration

```
device brake_pedal
features
        brake_event: out event data port float_type;
flows
        Flow1: flow source brake_event;
end brake_pedal;
--
system cruise_control
features
        brake_event: in event data port;
```

Table 10-1: Flow Declarations within a Component Type Declaration (cont.)

```
        throttle_setting: out data port float_type;
flows
        brake_flow: flow path brake_event -> throttle_setting ;
end cruise_control;
--
device throttle_actuator
features
        throttle_setting: in data port float_type;
flows
        Flow1: flow sink throttle_setting;
end throttle_actuator;
```


10.2 Flow Paths

Within a component **implementation**, **flow** declarations define the details of

- flow paths through a component
- end-to-end flows within the component

10.2.1 Flow Path through a Component

A **flow path** through a component consists of alternating sequences of paths through and connections among subcomponents within the component. This **path** begins and ends at features of the component type and is a realization of the corresponding **flow path** declared in the component's type declaration. Table 10-2 shows the **flows implementation** declarations through the component cruise_control.impl for the **flow path** brake_flow declared in the type declaration cruise_control of Table 10-1. It also shows a graphical representation of the **flow path**.

The **flows implementation** originates at the brake_event **event data port** and proceeds through to the **data port** throttle_setting. The **flow** involves the connections C1, C3, and C5 within the component **implementation** cruise_control.impl, as well as the paths through the subcomponents of that **implementation**. Notice that the nature of the **data** within the **flow** changes and involves **event data ports** as well as **data ports**.

Table 10-2:   Flow Implementation Declarations through a Component

```
system implementation cruise_control.impl
subcomponents
data_in: process interface;
control_laws: process control;
connections
C1: event data port brake_event -> data_in.brake_event;
C3: data port data_in.out_port -> control_laws.in_port;
C5: data port control_laws.out_port -> throttle_setting;
flows
brake_flow: flow path brake_event -> C1 -> data_in.interface_flow1 ->
                            C3 -> control_laws.control_flow1 -> C5 ->
throttle_setting;
end cruise_control.impl;
--
process interface
features
brake_event: in event data port ;
out_port: out data port float_type;
flows
interface_flow1: flow path brake_event -> out_port;
end interface;
--
process control
features
in_port: in data port float_type;
out_port: out data port float_type;
flows
control_flow1: flow path in_port -> out_port;
end control;
```
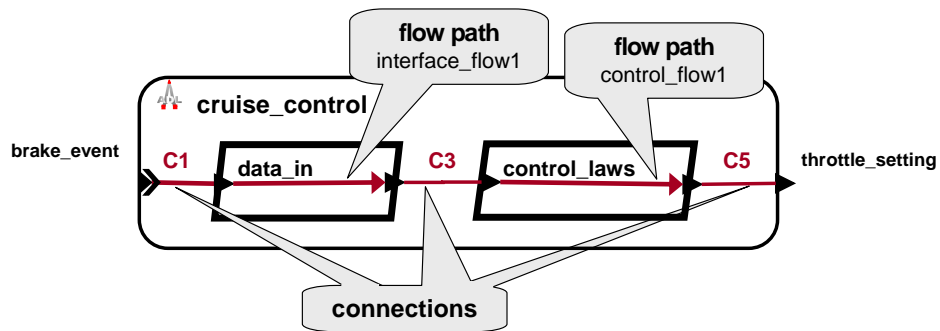

10.2.2 End-to-End Flow within a Component

An end-to-end flow within a component involves the declaration of a **path** from a flow **source** to a flow **sink** within the component. The partial specification in Table 10-3 illustrates this type of declaration: an end-to-end flow is defined between the **source** Flow1 in the **device** component brake_pedal and the **sink** Flow1 in the **device** component throttle_actuator.

Table 10-3:   An End-to-End Flow

```
system implementation complete.impl
subcomponents
brake_pedal: device brake_pedal;
cruise_control: system cruise_control;
throttle_actuator: device throttle_actuator;
connections
C1: event data port brake_pedal.brake_event ->
cruise_control.brake_event;
C2: data port cruise_control.throttle_setting ->
throttle_actuator.throttle_setting;
flows
brake_flow: end to end flow brake_pedal.Flow1 -> C1 ->
cruise_control.brake_flow -> C2 -> throttle_actuator.Flow1;
end complete.impl;
--

device brake_pedal
features
brake_event: out event data port;
flows
Flow1: flow source brake_event;
end brake_pedal;
--
system cruise_control
features
brake_event: in event data port;
throttle_setting: out data port float_type;
flows
brake_flow: flow path brake_event -> throttle_setting;
end cruise_control;
--
device throttle_actuator
features
throttle_setting: in data port float_type;
flows
Flow1: flow sink throttle_setting;
end throttle_actuator;
--
data float_type
end float_type;
```
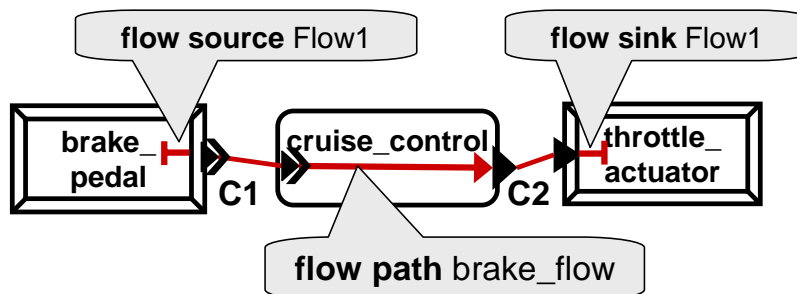
11 Properties

Properties provide descriptive information about

- components
- subcomponents
- features
- connections
- flows
- modes
- subprogram calls

A **property** has a name, type, and an associated value. **Properties** can be assigned values through **property** association declarations.

There are built-in **property** types and predeclared **properties** in the AADL standard. Collectively, these **properties** and **property** types encompass common attributes for the elements of the language. For example, a predeclared **property** of a **port** is Required_Connection, which is of type **aadlboolean** and has a value of *true* or *false*.[23] Its predeclared (default) value is *true*. However, a **property** association can assign the value *false*, allowing the **port** to be unconnected. A summary of AADL built-in **property** types is included on page 122 in the Appendix.

In addition to providing predeclared **properties** and built-in **property** types, AADL also permits the defining of new **properties** and **property** types. For example, to define a new **property** (e.g., Priority) for a **thread**, a user would declare a **property** name, type, and association of the new property. The **property** type declared for a new **property** may be a built-in type (e.g., **aadlinteger**), or a new type can be declared using a **property** type declaration.

11.1 Property Declarations

The declarations relating to **properties** are listed below.

- property association (Section 11.2): assigns a value or list of values to a named **property**

---

[23] Required_Connection is included in the predeclared property set named AADL_Properties that is part of every AADL specification [SAE 06a].

- property set (Section 11.3): defines a named collection of **property** types, names, and constants

- Property type (Section 11.4) defines a **property** type and specifies the set of acceptable values for **properties** of that type.

- Property name (Section11.5) defines a **property** by declaring a name, identifying a type for the **property**, and applying it to a category of element within the specification (i.e., mode, port group, flow, port, server subprogram, or connection).

- Property constant (Section 11.6) defines a name for a property value that can be referenced in **property** expressions wherever the value itself is permissible.

Property name, property type, and property constant declarations must be contained within a **property set** declaration.

11.2 Assigning Property Values

A **property** can be assigned a value or a list of values through a **property** association declaration. Property values can be associated with **properties** directly within individual component declarations, through an inherited value or an explicit contained **property** association referencing elements within a hierarchal component. In addition, **property** associations can be declared as being mode- or platform-binding specific.

11.2.1 Basic Property Associations

Property associations can be included within the **properties** section of component type or **implementation** declarations or within declarations for **subcomponents**, **features**, **connections**, **flows**, **modes**, and **subprogram calls** and their refinements.

Sample component **property** association declarations are shown in Table 11-1 where an **implementation** speed_data of the **thread** type data_processing is declared with associations for two standard **properties**. The Period **property** is assigned a single value of 100 ms. The Compute_Execution_Time assigned value is a range. In addition, the **in data port** declaration sensor_data includes a **property** association that declares the **port** need not be connected, and the **thread** subcomponent declaration for data_processing includes a **property** association declaring the initialization execution time range for the **thread** (1 ms .. 2 ms).

Table 11-1:   Basic Property Association Declarations

```
thread data_processing
features
sensor_data: in data port {Required_Connection => false;};
end data_processing;
--
thread implementation data_processing.speed_data
```

Table 11–1:   Basic Property Association Declarations (cont.)

```
properties
    Period => 100 ms;
    Compute_Execution_Time  => 5 ms .. 10 ms;
end data_processing.speed_data;
--
process implementation control.impl
subcomponents
data_processing: thread data_processing.speed_data
{Initialize_Execution_Time => 1 ms .. 2 ms;};
end control.impl;
```

Access property associations are used to detail the character of subcomponent **access**, both **requires** and **provides**. Table 11-2 shows two **access property** associations, where the **process** control **requires** read_only access to set point data data_sets.set_points and **provides** read_write access to its internal error logs. This is a modification of an example from Table 8-7.

Table 11-2:   Sample Access Property Associations

```
process control
features
cc_set_point_data: requires data access data_sets.set_points
                            {Required_Access => access read_only;};
error_log_data: provides data access logs.error_logs
                            {Provided_Access => access
read_write;};
end control;
```

11.2.2 Contained Property Associations

Property associations for individual components have been shown in earlier examples (e.g., Table 11-1). These declarations assign values for instances of the component. However, explicit **property** associations may be omitted for a number of the elements of an individual component. In these cases, values can be assigned through contained **property** association declarations or inherited from declarations higher in the component containment hierarchy.

A contained **property** association can be used to assign a property value to **subcomponents**, **features**, **flows**, **connections**, or **modes** defined within a component. A value can be assigned to an element that is deeply nested within the component. In addition, with contained **property** associations, configuration parameters for a **system** can be defined at a single point (e.g., at the highest point possible in the component hierarchy). In that way, the parameters provide a centralized set of **properties** and values for elements of a model that can readily be identified, adjusted, and reviewed.

An explicit contained **property** association is declared using an **applies to** clause that specifically identifies an element within the component. The identification **path** to the element consists of a dot-separated sequence of zero or more subcomponent identifiers

---

followed by the identifier of the **subcomponents**, **features**, **flows**, **connections**, or **modes** identifier to which the **property** association applies. Consider the partial specification in Table 11-3 that shows the relevant type and **implementation** declarations for a simplified cruise control system. The **property** associations within the **system implementation** declaration for cc_complete.impl are **property** associations for the execution time for the compute entry point of a contained **thread** control_algorithm and the required connection value for a **data port** of the contained **thread** adjust.

Table 11-3 shows two contained **property** associations within the **system implementation** cruise_control.impl. In the first association, the computation time for the compute entry point of the subcomponent **thread** control_algorithm is assigned the range of 2 ms.. 5 ms. The **thread** control_algorithm is contained within the **process** control_laws that is a subcomponent of the **system** cruise_control. In the second association, the Required_Connection **property** is assigned the value **false** for the **out data port** of the contained **thread** adjust.

Table 11-3:   Contained Property Associations

```
system cc_complete
properties
Period => 20ms;
end cc_complete;
--
system implementation cc_complete.impl
subcomponents
brake_pedal: device brake_pedal;
cruise_control: system cruise_control.impl;
throttle_actuator: device throttle_actuator;
connections
C1: event data port brake_pedal.brake_event ->
cruise_control.brake_event;
C2: data port cruise_control.throttle_setting ->
throttle_actuator.throttle_setting;
properties
Compute_Execution_Time => 2 ms.. 5 ms applies to
    cruise_control.control_laws.control_algorithm;

Required_Connection => false applies to
    cruise_control.control_laws.adjust.out_port;
end cc_complete.impl;
--
system implementation cruise_control.impl
subcomponents
data_in: process interface;
control_laws: process control.impl;
connections
C1: event data port brake_event -> data_in.brake_event;
C3: data port data_in.out_port -> control_laws.in_port;
C5: data port control_laws.out_port -> throttle_setting;
end cruise_control.impl;
--
process control
```

Table 11−3:   Contained Property Associations (cont.)

```
features
in_port: in data port ;
out_port: out data port ;
end control;
--
process implementation control.impl
subcomponents
adjust: thread adjust_sensor_value.impl;
control_algorithm: thread algorithm.impl;
end control.impl;
--
thread adjust_sensor_value
features
in_port: in data port;
out_port: out data port;
end adjust_sensor_value;
--
thread implementation adjust_sensor_value.impl
end adjust_sensor_value.impl;
--
thread algorithm
features
in_port: in data port;
out_port: out data port;
end algorithm;
--
thread implementation algorithm.impl
end algorithm.impl;
```
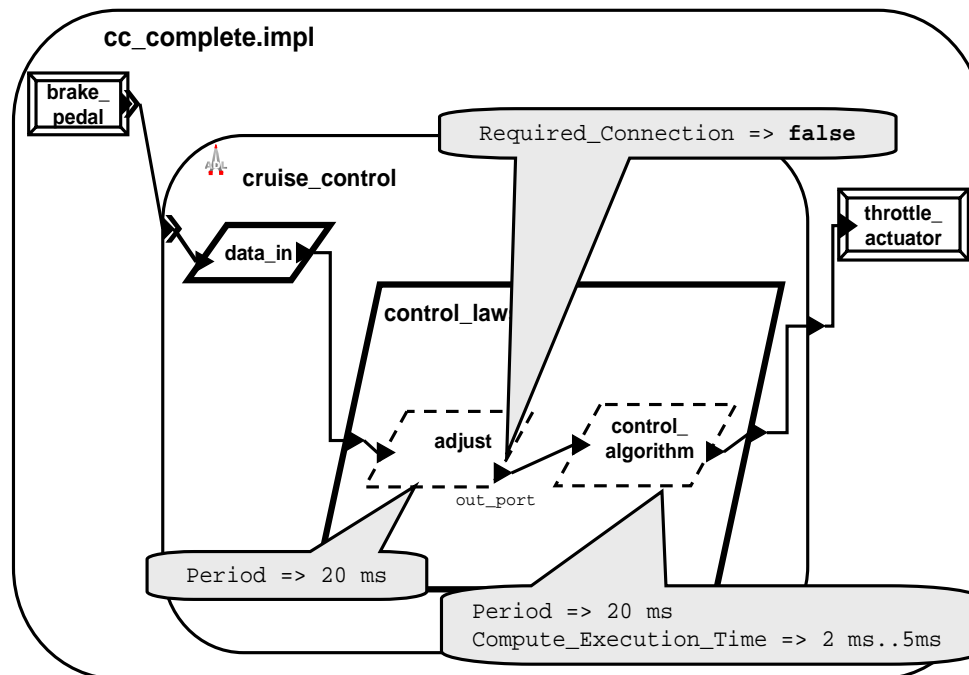
Contained **property** associations are required when a property value involves a reference to another part of a model. For example, the **binding property** of a **thread** must refer to the **processor** to which it is bound. However, that reference is represented as a path relative to the location at which the **property** association is specified. Thus, the **property** association must be declared as contained **property** association attached to a model component that is the common parent of the component being referenced and the component to which the property value belongs.

An example of a contained **property** association across a component hierarchy is shown in Figure 11-1 for the **property** Allowed_Processor_Binding. The **property** association is included in the specification for the **system** component Avionics_sys and declares that the **thread** observe can be bound to the **processor** linux1.
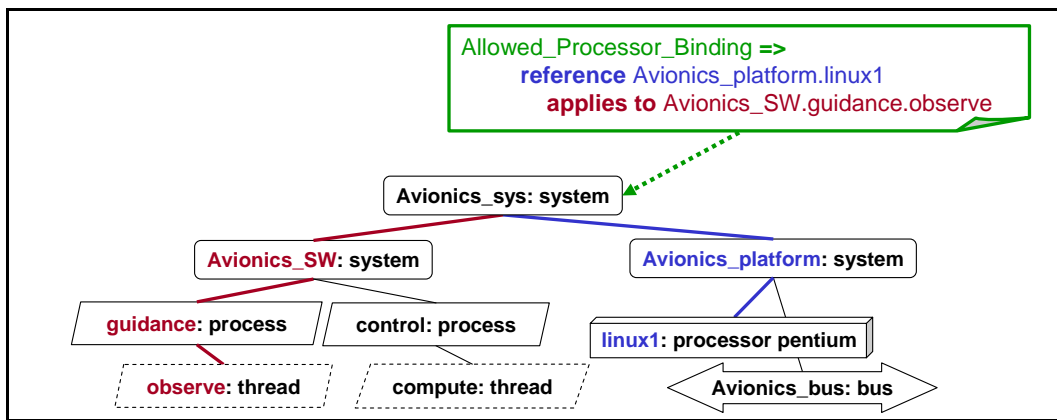
Figure 11-1:   Contained Property: Allowed_Processor_Binding

11.2.3 Inherited Property Associations

There is an implicit form of a **property** association that can be declared for contained components. This form involves **properties** defined with the **inherit** reserved word. For these **properties**, a **property** association declaration within a component is assigned to any subcomponent to which the **property** applies. For example, a Period **property** association within a **process** declaration applies to all of the threads contained within it, unless an individual **thread property** association declaration assigns a different value to the Period. An example Period **property** declaration within a **system** type declaration is shown in Table 11-3. A graphical representation is shown in the lower portion of that table. See Section 11.5 for more information.

One should be cautious in using this implicit **property** assignment for subcomponents. An inadvertent omission of a specific assignment for a contained component is not readily detectable and may result in an incorrect property value assignment. In the example shown in Table 11-3, Period for the **thread** adjust defaults to an execution time of 20 ms. If the intention had been to have a Period of 10 ms, there would have to be an explicit declaration for the Period of the adjust subcomponent**.**

11.2.4 Mode or Binding Specific Property Associations

Property associations can be specialized to specific **modes** or bindings by declaring this specialization in the **property** association. For example, the computation time and period **property** associations from Table 11-3 are declared for a specific **processor binding** in Table 11-4. Thus, alternative **thread** execution times and other processor-dependent **properties** can be declared based upon **processor** bindings through the **in binding** declaration. In Table 11-4, the Required_Connection **property** association is specialized to the initialize **mode** in the **system implementation** declaration cc_complete.impl.

Table 11-4:   In Binding and In Mode Property Associations

```
system cc_complete
properties
Period => 20 ms in binding (Intel);
end cc_complete;
--
system implementation cc_complete.impl
subcomponents
brake_pedal: device brake_pedal;
cruise_control: system cruise_control.impl;
throttle_actuator: device throttle_actuator;
Intel: processor Intel.impl;
connections
C1: event data port brake_pedal.brake_event ->
cruise_control.brake_event;
C2: data port cruise_control.throttle_setting ->
throttle_actuator.throttle_setting;
modes
initialize: initial mode;
nominal: mode;
properties
Compute_Execution_Time => 2 ms.. 5ms applies to
cruise_control.control_laws.control_algorithm in binding (Intel);
Compute_Execution_Time => 3 ms.. 7ms applies to
cruise_control.control_laws.control_algorithm in binding (AMD);
Required_Connection => false applies to
  cruise_control.control_laws.adjust.out_port in modes (initialize);
end cc_complete.impl;
--
processor Intel
end Intel;
--
processor implementation Intel.impl
end Intel.impl;
```

11.2.5 Property Values

The values that are assigned to **properties** can take a variety of forms:

- individual values associated with a basic built-in type like **aadlboolean**, **aadlstring**, **aadlinteger**, or **aadlreal**

- a **range** of values, as shown in Table 11-4 for execution times

- values with or without units (e.g., Period)

- an **enumeration** value set (e.g., the Required_Access **property**)

- values that include model elements as well as explicit component classifiers

- individual values or lists of values

The **property** type **reference** allows a property value to refer to a model element according to the containment hierarchy. For example, in Figure 11-1 the Allowed_Processor_Binding declaration references a specific **processor** in the **system** hierarchy. The **properties** of type **classifier** allow component classifiers to be used as property values. In Table 11-5, the first **property** association for the **property** Allowed_Processor_Binding_Class restricts the **binding** to **processors** of type PowerPC. The **classifier** value can be a component **implementation** or a list of **classifier** references, as shown in the second **property** association for the **property** Allowed_Processor_Binding_Class in the lower part of Table 11-5.

Table 11-5:   Classifier Property Types

```
Allowed_Processor_Binding_Class => processor PowerPC;
--
processor PowerPC
end PowerPC;
--
Allowed_Processor_Binding_Class => (processor PowerPC.G4, processor
PowerPC.G5);
-- where PowerPC.G4 and PowerPC.G5 are processor implementations of
-- of the processor type PowerPC
```

Property value assignments can be indirect and used to centralize the declarations of system parameters. For example, the **property** associations in Table 11-6 use the keyword **value** to assign values to the Deadline and Period **properties** of the **thread** algorithm.impl. In the **property set** timing, the **property** HiRate is defined as a **constant** of the type Time with a value of 5 ms. Period is assigned the value of HiRate, and the Deadline is assigned the value of Period. Thus, a change in all of these assignments can be accomplished simply by changing the value of HiRate.

Table 11-6:   Property Associations with Value

```
thread implementation algorithm.impl
properties
Deadline => value (Period);
Period => value (timing::HiRate);
end algorithm.impl;
--
property set timing is
HiRate: constant Time => 5 ms;
end timing;
```

Built-in **property** types are summarized on page 122 in the Appendix. Details on declaring additional **property** types are discussed in Section 11.4.

11.3 Defining New Properties

A property set is a named collection of property type, property name, and property constant declarations. A named **property set** can be used to augment a general specification or defined as part of an AADL **annex**.

Table 11-7 shows the form and content of a sample **property set** declaration set_of_faults and includes examples of property name, property type, and property constant declarations. The **property** named comm_error_status is defined as a **property** of type **aadlboolean** (**true** or **false**) that applies to **system** and **device** components. A **property** type Speed_Range is defined as a range of real values from 0.0 mph..150.0 mph. The **constant** Maximum_Faults is defined as the integer value 3.

For more details on

- property type declaration: see Section 11.4
- property name declaration: see Section 11.5
- property constant declaration: see Section 11.6

Table 11-7:   Sample Property Set Declarations

```
system implementation data_processing.accelerometer_data
properties
      set_of_faults::comm_error_status => true;
end data_processing.accelerometer_data;

property set set_of_faults is

-- An example property name declaration
comm_error_status: aadlboolean applies to (system, device);
-- An example property type declaration
Speed_Range : type range of aadlreal 0.0 mph..150.0 mph units (mph);
-- An example property constant declaration
Maximum_Faults : constant aadlinteger => 3;

end set_of_faults;
```

11.4 Property Type Declarations

A property type declaration defines a property type by associating an identifier with it and establishing the set of legal values for a property of that type. The declaration consists of

1.   the desired identifier for the property type

2.   a colon (:)

3.   the reserved word **type**

4.   an explicit type definition

5.   a terminating semicolon (;)

The pattern for a property type declaration is shown in the box below:

identifier: **type** *property type definition*;

A property type definition may be an AADL built-in property type, a specialized type explicitly defined within the declaration, or a reference to previously defined property type.

In the examples shown in Table 11-8, the **property** type bit_error is defined as an **aadlboolean** property type. The predefined **aadlboolean** property type has two legal values, **true** and **false**. The property types fault_category and fault_condition are defined as **enumeration** types. An **enumeration** property type defines a specific set of identifiers as its legal values.

Type declarations can be more complex than simple base types. For example, the type number_of_components is declared in the **property set** more_types as an aadlinteger that ranges over the value 0 .. 25. The **property** boat_length is declared as a type of **aadlreal** with the units of feet that ranges over the values of 7.5

`.. 150.0` **units** `( feet )`. The **property** `voltage_ranges` is a type of **aadlreal** that is a range of values that can span `0.0 .. 5.3` **units** `(volts)`.

Table 11-8:   Sample Property Type Declarations

```
property set set_of_faults is
bit_error: type aadlboolean;
fault_category: type enumeration (benign, tolerated, catastrophic);
fault_condition: type enumeration (okay, error, failed);
time_delay: type aadlreal units (seconds) ;
end set_of_faults;

property set more_types is
number_of_components: type aadlinteger 0 .. 25;
boat_length : type aadlreal 7.5 .. 150.0 units ( feet );
voltage_ranges : type range of aadlreal 0.0 .. 5.3 units (volts);
end more_types;
```

11.5 Property Name Declarations

A property name declaration defines a property by declaring a name, identifying a type for the property, and applying the property to a category of element within the specification (i.e., component, mode, port group, flow, port, server subprogram, or connection). A property name declaration consists of

1. desired identifier for the property name

2. colon (:)

3. neither, either, or both of the reserved words (**access** or **inherit**)

4. explicit type identifier

5. reserved words (**applies to**)

6. property owner category or the reserved word (**all**)

7. terminating semicolon (;)

The pattern for a property name declaration is shown in the box below:

name : **[access inherit** property type **applies to** (property owner category);

A property owner category can be a component (e.g., system, thread, device), mode, port group, flow, port (event or data), server subprogram, parameter, or connections (port group, event port, data port, access, or parameter).

Example property name declarations within a **property set** set_of_names are shown in Table 11-9. Property name declarations can include the **access** and **inherit** options. A **property** declared with the reserved word **inherit** indicates that a value is inherited from a containing component, if a property value cannot be determined for a component.

This inheritance can be seen in the declaration for the **property** `critical_unit` that is declared as **inherit** and as type **aadlboolean** and applies to all component categories. A **property** declared with the reserved word **access** is associated with access to a subcomponent rather than to the data component itself. The **property** `queue_access` is declared as a true-false **access property** for a data component. This can be used to restrict required access to a data queue. The **property** `required_sensor_array_size` is declared as type `array` that is declared within the **property set** `set_of_types` that is shown in the lower portion of Table 11-9. Similarly, the **property** `dangerous_voltages` is declared with a type `voltage_ranges` that is declared in the **property set** `more_types` found in Table 11-8.

Table 11-9:   Sample Property Name Declarations

```
property set set_of_names is
critical_unit: inherit aadlboolean applies to (all);
queue_access: access aadlboolean applies to (data);
required_sensor_array_size: inherit set_of_types::array applies to
(system, process, thread);
dangerous_voltages: more_types::voltage_ranges => 5.1 .. 5.3 volts
applies to (processor);
end set_of_names;

property set set_of_types is
array: type enumeration (single, double, triplex);
end set_of_types;
```

11.6 Property Constant Declarations

Property constants are property values that are known by a symbolic name. Property constants are provided in the predeclared property sets and can be defined in named property sets. They can be referenced in **property** expressions by name wherever the value itself is permissible.

Here are the basic declaration forms for a property constant declaration:

identifier: **constant** (**type**) => property value
identifier: **constant list of** (**type**) => property values

In the forms shown above

- *Identifier* is the name that can be used as a value in **property** associations.

- Entry (*type*) is a built-in type or a type declared in a **property set**.

- Property value or values must be of the type included in the *constant* declaration.

Some sample declarations are shown in Table 11-10, where, for the **property set** limits_set,

- Max_Threads is defined as an integer value of 256.

- Minimum_value is defined as a real value of 5.0.

- Default_Fault_State is defined as a constant of the type fault_condition with the value of okay.

The type fault_condition, mentioned in Table 11-10, is defined in the **package** set_of_faults, as shown in Table 11-8.

Table 11-10:   Sample Property Constant Declarations

```
property set limits_set is
Max_Threads : constant aadlinteger => 256 ;
Minimum_value: constant aadlreal => 5.0;
Default_Fault_State: constant set_of_faults::fault_condition =>
okay;
end limits_set
```

12 Organizing a Specification

This section presents language constructs that can be used to organize an AADL specification by grouping like elements using packages or design patterns.

12.1 Packages

A **package** is a named grouping of declarations and **property** associations that can be used to organize a specification. Packages establish distinct namespaces. However, they do not define an architectural hierarchy or design structure and cannot be declared inside other packages.

A **package** is divided into **public** and **private** segments. Declarations in the **public** segment are visible outside the **package**, whereas declarations in the **private** segment are visible only within the **package**. To reference an element in the **public** segment from outside a **package**, preface the element's identifier with the **package** name. In Table 12-1 for example, a **process** type compress_display_data contained in the **public** segment of the **package** display_dynamics_set would be referenced from outside the **package** as display_dynamics_set::compress_display_data.

Also in Table 12-1, the specification for the **system** display_management references the compress_display_data **process** declared in the **package** display_dynamics_set. The **data** component new_format declared in the **private** segment of the **package** cannot be accessed from outside. However, the **data** component display_data can be, since it is declared in the **public** segment of the **package**.

Table 12-1:   Example Package Declaration

```
package display_dynamics_set
-- Elements accessible from outside the package are listed following
-- the key word public
public
process compress_display_data
features
display_data_input: in data port display_data;
formatted_data: out data port;
data_error: out event port;
end compress_display_data;

data display_data
end display_data;
-- Elements accessible only inside the package are listed following
-- the key word private
private
data new_format
end new_format;
end display_dynamics_set;

-- The subcomponent declaration below references a process in
-- display_dynamics_set
system implementation display_management.impl
subcomponents
compress_data: process display_dynamics_set::compress_display_data;
….
end display_management.impl;
```

A package name can include multiple identifiers separated by a double colon (::). Thus, a package name like "primary_control_system::roll_axis::control_components" is permitted. This naming flexibility can be useful for packages that have been developed independently and have been assigned the same name. For example, consider two engineering teams working on a project, *team red* and *team blue*. Each team develops a package with the name "sensor_control." These packages can be renamed "team_red::sensor_control" and "team_blue::sensor_control".[24] This would establish separate namespaces for each package and allow references to components with the same name within each package. That is, "team_red::sensor_control::controller" would reference a different declaration than "team_blue::sensor_control::controller." In addition, this flexibility can be used to associate packages logically. For example, two packages "roll_control" and "yaw_control" can be associated by renaming them "aircraft::roll_control" and "aircraft::yaw_control."

Packages can be used to organize layers of a design. For example, a **package** can be defined for a flight manager subsystem using constituent component subsystems, packages

---

[24]   The AADL standard states that "A defining package name must be unique in the global namespace. This means that the first identifier in a package name must be unique in the global namespace. Succeeding identifiers in the package name must be unique within the scope of the previous identifier" [SAE 06a].

that contain generic (common) descriptions, or packages containing only data types (e.g., a data dictionary). This concept is shown in the partial specification and packages of Table 12-2 where the `Flight_Manager` type declaration and declarations within the **package** `avionics_subsystems` reference components defined in separate packages.

In particular, in the portion of Table 12-2 labeled ①, the `Flight_Manager` component type declaration extends the `Flight_Manager` system type declared in the `avionics_subsystems` **package**. In the section labeled ②, the data type `avionics_data::raw_data`, declared in the **package** `avionics_data` in the section labeled ④, is used in the `avionics_subsystem` **package**. And, in table section ③, the GPS subcomponent is an instance of the **implementation** `GPS.impl` from the `avionics_sensor` **package**. The comment lines (`--` ......) indicate that other declarations required for a complete system specification are not shown.

Table 12-2:   Example Design Organization Using Packages

```
system Flight_Manager                                                        ①
extends avionics_subsystems::Flight_Manager
end Flight_Manager ;
--
system implementation Flight_Manager.common
    subcomponents
       NSP : process avionics_subsystems::NavigationSensorProcessing;
       GPS : device  avionics_sensors::GPS.mil;
-- ………
end Flight_Manager.common;
```

```
package avionics_subsystems                                                  ②
public
  system Flight_Manager
  features
input_data: in data port avionics_data:: raw_data;
output_data: out data port avionics_data:: processed_data;
  end Flight_Manager ;
--
process NavigationSensorProcessing
end NavigationSensorProcessing;
-- ……
end avionics_subsystems ;
```

```
package avionics_sensors                                                     ③
public
  device GPS
  end  GPS;
  --
  device implementation GPS.mil
  end GPS.mil;
--………
end avionics_sensors;
```

Table 12-2: Design Organization Using Packages   (cont.)

```
package avionics_data                                          ④
public
--
data raw_data
end raw_data;
--
data processed_data
end processed_data;
--
end avionics_data;
```

12.2 Design Patterns

A collection of specifications can be defined that form a set of extensible design patterns. Using AADL extension and refinement capabilities, these patterns can be used to develop specific application models.

12.2.1 Type Extensions

Elements of a design pattern set can involve core type declarations whose **features** are only partially defined. These core types as well as their descendents can be repeatedly extended, defining more specific types through feature refinements (**refined to**), as shown in Table 12-3. In that example, the core type one_dimensional_control is extended to form two specific types: (1) roll_control and (2) pitch_control. In these extensions, the partially defined **in port** and **out port** are **refined to** include specific data types. For the type declaration for roll_control, another input **data port** is added.

In general, new **features** can be added; partially defined **features**, completed; and **property** associations, added or modified. In the example in Table 12-3, the Required_Connection property value is changed in the roll_control extension.[25] In the pitch_control extension, the Source_Name **property** association is added. The refinement options for type extension declarations are summarized on page 124 in the Appendix.

---

[25] The default value for the predeclared property Required_Connection is *true*. However, it is declared explicitly as true in this example to demonstrate the refinement of property associations.

Table 12-3:   Example Type Extension

```
process one_dimensional_control
features
commanded_value: in data port;
actuator_command: out data port {Required_Connection => true;};
end one_dimensional_control;


process roll_control extends one_dimensional_control
features
commanded_value: refined to in data port  roll_cmd_data;
actuator_command: refined to out data port aileron_cmd_data
                                      {Required_Connection =>
false;};
cross_coupling_state: in data port coupling_data;
end roll_control;
process pitch_control extends one_dimensional_control
features
commanded_value: refined to in data port pitch_cmd_data
                         {Source_Name => "commanded_pitch_file";};
actuator_command: refined to out data port elevator_cmd_data;
end pitch_control;
```

12.2.2 Refinements within Implementations

In an **implementation** declaration, the **refines type** subclause can be used to add or modify feature **property** associations of an implementation's type. For example, consider the **server subprogram features** for the **thread** type reader shown graphically and as AADL text in Table 12-4. There are two **thread implementations**, one for reading temperature (reader.temp) and one for reading pressure (reader.pressure). Each modifies the computation execution time value and adds a **property** association that defines a value for the subprogram's compute deadline. Note that including the name of the feature being refined (in this example a **subprogram**) in the **refined to** statement is optional. In the example, the **subprogram** read_data is included within the **refined to** declaration for the **thread implementation** reader.temp but is not included in the **refined to** declaration for the **thread implementation** reader.pressure.

Table 12-4:   Example Refines Type Implementation Subclauses

```
thread reader
features
read_it: server subprogram read_data {Compute_Execution_Time => 2
ms ..5 ms;};
end reader;

thread implementation reader.temp
refines type
read_it: refined to server subprogram read_data
{Compute_Execution_Time => 2 ms .. 4 ms; Compute_Deadline => 5 ms;
};
end reader.temp;

thread implementation reader.pressure
refines type
read_it: refined to server subprogram {Compute_Execution_Time => 2
ms .. 4 ms;
        Compute_Deadline => 5 ms; };
end reader.pressure;
```
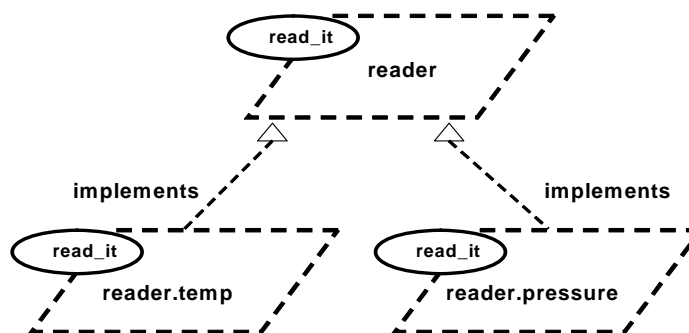

12.2.3 Implementation Extensions

Implementations can extend other implementations, modifying the underlying implementations and adding characteristics to them. Individual implementations can be extended multiple times, and extensions themselves can be extended. **Implementation** extensions can be integrated with type extension declarations to create an interrelated set of component types and implementations.

Table 12-5 shows example **implementation** extension declarations with accompanying type extension declarations for a flight control system. The type extension for `flight_control_system` adds an additional **in data port** `sensor_set_redundant`. Relationships among the declarations are shown graphically following the textual AADL specification. The refinement options for **implementation** extension declarations are summarized on page 125 in the Appendix.

Table 12-5:   Example Implementation Extensions

```
system flight_control_system
features
sensor_set: in data port;
end flight_control_system;

system flight_control_system_redundant extends
              flight_control_system
features
sensor_set_redundant: in data port;
end flight_control_system_redundant;

system implementation flight_control_system.basic
end flight_control_system.basic;

system implementation flight_control_system.UAV extends
              flight_control_system.basic
end flight_control_system.UAV;

system implementation flight_control_system_redundant.R_UAV extends
flight_control_system.UAV
end flight_control_system_redundant.R_UAV;
```
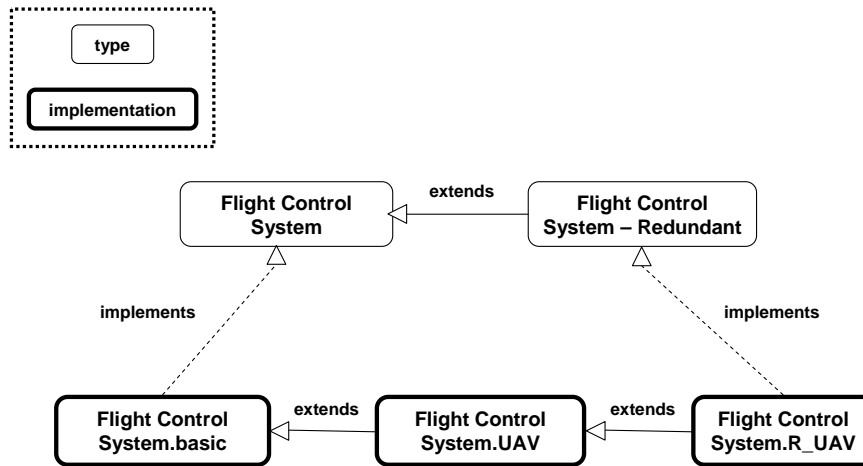
12.2.4 Example Design Patterns

In this section, the extension and refinement capabilities of the AADL are used to define a family of N-way Voting Lane components. Each N-way component constitutes a lane within a redundant composite of N-lanes and receives output data and system status opinions from the other lanes. Figure 12-1 shows a three-way lane **system** component.
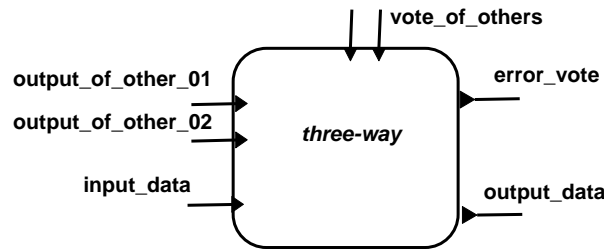
Figure 12-1:   Three-way Voting Lane Component

The family of N-way Lane components depicted in Figure 12-2 is built upon extensions and refinements of generic **type**-**implementation** pairs. The core pair is a two-way voting generic type two-way and a generic **implementation** of that type two-way.g. The generic two-way voting type and **implementation** are extended to create a three-way voting generic **type**-**implementation** pair; the generic three-way voting type and **implementation** are extended to create a four-way voting generic **type**-**implementation** pair.
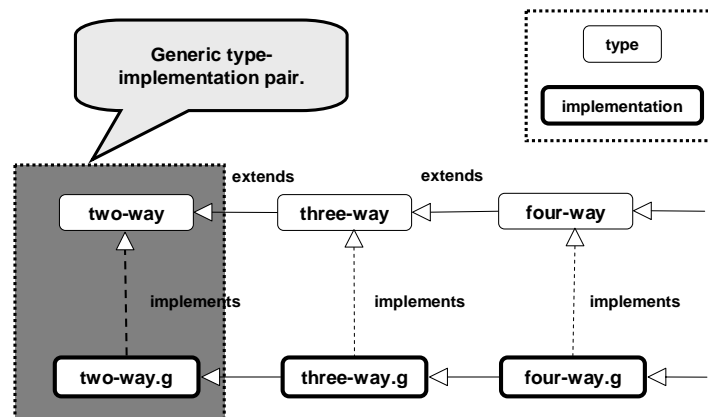
Figure 12-2:   Generic N-way Voting Lanes Type-Implementation Pairs

Generic **type**-**implementation** pairs can be extensions along a well-defined aspect of the design. In this example of N-way lane components, the aspect is the number of redundant lanes (voting ways) for the **system**. Generic implementations consist of general subclause declarations that can readily be refined in subsequent **implementation** extensions. In cascading generic implementations, partially defined **subcomponents**, **calls**,

connections, **flows**, and **modes** are added. As appropriate, **property** associations are modified and added.

In cascading generic type declarations, **features** are partially defined, and basic **property** associations are declared. Generic type declarations consist of the following elements:

- partially defined **features** that can be completed in the refinements of a specialized **extends** type declarations

- basic flow declarations that can be used throughout the family with modifications only to the flow declaration **property** associations

- general **property** associations that characterize a component

In creating the family of **type**-**implementation** pairs illustrated in Figure 12-2, for instance, the two-way generic type is extended to create a three-way type by adding **features** that are partially defined rather than complete (e.g., **data ports** without data **classifiers** to handle the additional inputs from other lanes). The three-way generic **implementation** results from the extension of the two-way generic **implementation**. In this **implementation** extension, **subcomponents**, **connections**, **modes**, and other elements are added. Generic declarations should be sufficiently general to allow refinement by subsequent "voting" **implementation** extensions. The extension and refinement capabilities for types and implementations are summarized on pages 124–125 in the Appendix.

A specific realization of an aspect (e.g., a three-way system) is defined by an extension of the associated **type**-**implementation** pair, as shown in Figure 12-3. In the specific type extensions (**extends**), **features** are completed, **features** and **flows** are added, and relevant **property** associations are modified or added.

These declarations result in specialized realizations of the generic type. The specific **implementation** extensions (such as the three-way **implementation** generated from the three-way.g **implementation** in Figure 12-3) refine the general pattern of their associated generic implementations, providing all of the details required for instantiation. In the extension **subcomponents** definitions are completed; and **calls**, **connections**, **flows**, and **modes** are added.
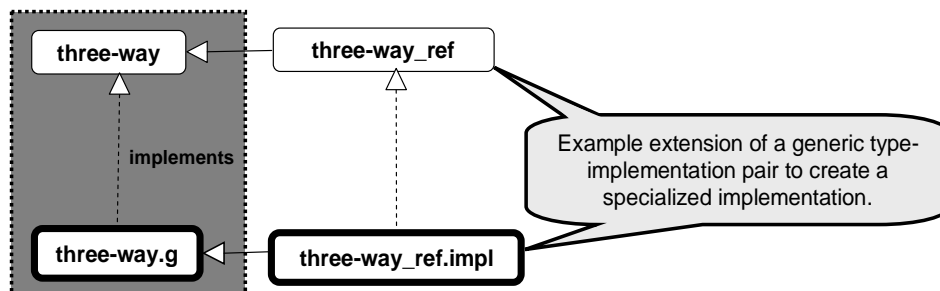
Figure 12-3:   Specialized Extension and Refinement

Appendix

Component-Subcomponent Relationships

Table 13-1 summarizes the permitted component–subcomponent relationships for each of the component abstractions in the AADL.

Table 13-1:   Allowed Component-Subcomponent Relationships

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

Allowed Features

Table 13-2 and Table 13-3 summarize the allowed features for each of the component abstractions in the AADL.

Table 13-2:   Allowed Features for Components

Category Group	Component Category	Allowed Features
Software	process	• server subprogram • port/port group • provides data access • requires data access
	thread	• server subprogram • port/port group • provides data access • requires data access
	data	• subprogram • provides data access
	thread group	• server subprogram • port/port group • provides data access • requires data access
	subprogram	• out event port • out event data port • port group (event only) • requires data access • parameter
Execution Platform	processor	• server subprogram • port/port group • requires bus access
	memory	requires bus access
	bus	requires bus access
	device	port/port group • server subprogram • requires bus access
Composite	system	• server subprogram • port/port group • provides data access • provides bus access • requires data access • requires bus access

Table 13-3:   Features and Allowed Components

Feature		Allowed Feature of Component or Component Category
port **port group**	all port types	• system • process • thread • thread group • processor • device
	• event port • event data port • port group (events only)	subprogram (component)
subprogram	server	• system • process • thread • thread group • processor • device
	subprogram (data)	data
access	provides data	• system • process • thread • thread group • data
	requires data	• system • process • thread • thread group • subprogram (component)
	provides bus	system
	requires bus	• system • processor • memory • bus • device
parameter		subprogram (component)

Constraints Summary

Table 13-4 contains a summary of the legality rules for AADL components from Version 1.0 of the standard.

Table 13-4:   Constraints/Restrictions for Components

Component Category	Type	Implementation
data	Features: • subprogram • provides data access Flow specifications: no Properties yes	Subcomponents: • data Subprogram calls: no Connections: access Flows: no Modes: yes Properties yes
subprogram	Features: • out event port • out event data port • port group • requires data access • parameter Flow specifications: yes Properties yes	Subcomponents: • none Subprogram calls: yes Connections: yes Flows: yes Modes: yes Properties yes
thread	Features: • server subprogram • port • provides data access • requires data access Flow specifications: yes Properties yes	Subcomponents: • data Subprogram calls: yes Connections: yes Flows: yes Modes: yes Properties yes
thread group	Features: • server subprogram • port • provides data access • requires data access Flow specifications: yes Properties yes	Subcomponents: • data • thread • thread group Subprogram calls: no Connections: yes Flows: yes Modes: yes Properties yes
process	Features: • server subprogram • port • provides data access • requires data access Flow specifications: yes Properties yes	Subcomponents: • data • thread • thread group Subprogram calls: no Connections: yes Flows: yes Modes: yes Properties yes

Table 13-4:   Constraints/Restrictions for Components (cont.)

Component Category	Type	Implementation
processor	Features: • server subprogram • port/port group • requires bus access Flow specifications: yes Properties yes	Subcomponents: • memory Subprogram calls: no Connections: no Flows: yes Modes: yes Properties: yes
memory	Features • requires bus access Flow specifications: no Properties yes	Subcomponents: • memory Subprogram calls: no Connections: no Flows: no Modes: yes Properties yes
bus	Features • requires bus access Flow specifications: no Properties yes	Subcomponents: • none Subprogram calls: no Connections: no Flows: no Modes: yes Properties yes
device	Features • port/port group • server subprogram • requires bus access Flow specifications: yes Properties yes	Subcomponents: • none Subprogram calls: no Connections: no Flows: yes Modes: yes Properties yes
system	Features: • server subprogram • port/port group • provides data access • provides bus access • requires data access • requires bus access Flow specifications: yes Properties yes	Subcomponents: • data • process • processor • memory • bus • device • system Subprogram calls: no Connections: yes Flows: yes Modes: yes Properties yes

Built-in Property Types

Table 13-5 summarizes the AADL standard built-in property types.

Table 13-5:   AADL Built-in Property Types

Property Type	Definition
aadlboolean	Two values, true or false
aadlstring	All legal strings of the AADL
enumeration	An explicitly listed set of enumeration identifiers as the set of legal values
units	An explicitly listed set of measurement unit identifiers as the set of legal values
aadlreal	A real value or a real value and its measurement unit
aadlinteger	An integer value or an integer value and its measurement unit
range	Closed intervals of numbers indicating that a property of this type has a value that is itself a range term and specifies the number type of values in the range term
classifier	Subset of syntactically legal component classifier references whose category matches one of component categories in the specified list
reference	Subset of syntactically legal references to those components, whose category matches one of component categories in the specified list, or to connections or to server subprogram features; indicated by the reserved word reference

AADL Reserved Words

Table 13-6 lists the AADL reserved words. Reserved words are case insensitive.

Table 13-6:   AADL Reserved Words

aadlboolean	end	modes	reference
aadlinteger	enumeration	none	refined
aadlreal	event	not	refines
aadlstring	extends	of	requires
access	false	or	server
all	features	out	set
and	flow	package	sink
annex	flows	parameter	source
applies	group	path	subcomponents
binding	implementation	port	subprogram
bus	In	private	system
calls	inherit	process	thread
classifier	initial	processor	to
connections	inverse	properties	true
constant	Is	property	type
data	list	provides	units
delta	memory	public	value
device	mode	range	

Refinements within Type Extensions

Table 13-7 summarizes the refinement capabilities within type extension declarations.

Table 13-7:   Type Extensions and Associated Refinements

Refinements within Type Extensions			
Subclause	**Refinement**		**Description (refined to)**
features	ports	data event data	• add ports (no **refined to**) • complete partial declaration (add a data type or an implementation classifier; change a data type classifier to a data implementation classifier) • redefine or add port property associations
		event	• add event ports (no **refined to**) • redefine or add event port property associations
	port group		• add port groups (no **refined to**) • complete partial declarations (add missing type reference; change data type classifier to implementation classifier) • redefine or add port group property associations
	subprogram		• add server or data subprogram features (no **refined to)** • complete partial declarations (change type classifier to an implementation classifier; no changes of subprogram type or implementation classifiers) • redefine or add subprogram property associations
	parameters		• add parameters (no **refined to**) • complete partial declaration (no change of parameter classifier to type or implementation; change a type classifier to implementation) • redefine or add parameters property associations
	subcomponent access		• add subcomponent access features (no **refined to**) • complete partial declaration (no subcomponent classifier to type or implementation; type classifier to implementation) • redefine or add subcomponent property associations
flows			• add flow specifications (no **refined to**) • redefine or add flow property associations
properties			• redefine or add component property associations

Refinements within Implementation Declarations

Table 13-8 summarizes the refinements associated within standard implementation declarations and implementation declarations that **extends** another.

Table 13-8:   Implementations Extensions and Associated Refinements

Refinements within Implementation Extensions	
Subclause	**Refinement Description**
refines type	• redefine or add feature property associations
subcomponents	• add subcomponents (no **refined to**) • complete partially referenced component classifier declaration • modify **in modes** with a new set of mode references • redefine or add subcomponent property associations
calls	• add calls or call sequences (no **refined to**) • no modification of call sequences
connections	• add connections (no **refined to**) • modify "in modes" references • redefine or add connection property associations
flows	• add flow specifications (no **refined to**) • modify **in modes** with a new set of mode references or mode transition references • redefine or add flow implementation property associations
modes	• add modes (no **refined to**) • redefine or add mode property associations
properties	• redefine or add component property associations

Index

A

B

C

D

E

Index

Index

References

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

[Feiler 04]   Feiler, P. H.; Gluch, D. P.; Hudak, J. J.; & Lewis, B. A. *Embedded System Architecture Using SAE AADL* (CMU/SEI-2004-TN-005). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2004. http://www.sei.cmu.edu/publications/documents /04.reports/04tn005.html

[SAE 06a]   Society of Automotive Engineers. *SAE Standards: Architecture Analysis & Design Language (AADL), AS5506, November 2004*. http://www.sae.org/servlets/productDetail?PROD_TYP=STD&PR OD_CD=AS5506 (2006)

[SAE 06b]   Society of Automotive Engineers. *SAE Standards for Works in Progress: Error Model Annex, Draft version 0.91*. http://www.sae.org/servlets/productDetail?PROD_TYP=STD&PR OD_CD=AS5506/2&HIER_CD=TEAAS2&WIP_SW=YES (2005)

[W3C 04]   World Wide Web Consortium (W3C). *Extensible Markup Language (XML) 1.0 (Third Edition)*. http://www.w3.org/TR/2004 /REC-xml-20040204/ (2004)

REPORT DOCUMENTATION PAGE

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	February 2006	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
The Architecture Analysis & Design Language (AADL): An Introduction	FA8721-05-C-0003

6. AUTHOR(S)

David P. Gluch, Peter H. Feiler, John J. Hudak

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2006-TN-011

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

In November 2004, the Society of Automotive Engineers (SAE) released the aerospace standard AS5506, named the Architecture Analysis & Design Language (AADL). The AADL is a modeling language that supports early and repeated analyses of a system's architecture with respect to performance-critical properties through an extendable notation, a tool framework, and precisely defined semantics.

The language employs formal modeling concepts for the description and analysis of application system architectures in terms of distinct components and their interactions. It includes abstractions of software, computational hardware, and system components for (*a*) specifying and analyzing real-time embedded and high dependability systems, complex systems of systems, and specialized performance capability systems and (*b*) mapping of software onto computational hardware elements.

The AADL is especially effective for model-based analysis and specification of complex real-time embedded systems. This technical note is an introduction to the concepts, language structure, and application of the AADL.

14. SUBJECT TERMS	15. NUMBER OF PAGES
AADL, architecture design language	144

16. PRICE CODE

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