SAE Aerospace
An SAE International Group

AEROSPACE STANDARD

SAE AS5506A

Issued 2004-11
Revised 2009-01

Superseding AS5506

(R) Architecture Analysis & Design Language (AADL)

RATIONALE

The language defined in SAE AS5506 has been refined and extended based on industrial experience with version 1.0 over the last 4 years. The improvements focus on better support for architecture templates and modeling of layered and partitioned architectures.

This Architecture Analysis & Design Language (AADL) standard document was prepared by the SAE AS-2C Architecture Description Language Subcommittee, Embedded Computing Systems Committee, Aerospace Avionics Systems Division.

TO PLACE A DOCUMENT ORDER: **Tel: 877-606-7323 (inside USA and Canada)**
Tel: 724-776-4970 (outside USA)
Fax: 724-776-0790
Email: custsvc@sae.org

SAE WEB ADDRESS: **http://www.sae.org**

SAE values your input. To provide feedback on this Technical Report, please visit
http://www.sae.org/technical/standards/AS5506A

TABLE OF CONTENTS

Table of Figures

Foreword

(1) The AADL standard was prepared by the SAE Avionics Systems Division (ASD) Embedded Computing Systems Committee (AS-2) Architecture Description Language (AS-2C) subcommittee.

(2) This standard addresses the requirements defined in SAE ARD 5296, Requirements for the Avionics Architecture Description Language[1].

(3) The AADL standard consists of a core language standard that is defined in this document and a collection of standardized property sets and/or sublanguages that are defined in annex documents. The core language standard provides full support for modeling the application task and communication architecture, the hardware platform, and the physical environment of embedded software-intensive systems, including standardized predeclared properties to characterize task execution and communication timing, as well as deployment of the application on the hardware platform. The standardized extensions allow core AADL models to be annotated with information that is not represented by the core language to meet specific embedded system analysis needs such as security analysis, dependability analysis, and behavioral analysis, and support for automated generation and integration of systems.

(4) The starting point for the AADL standard development was MetaH, an architecture description language and supporting toolset, developed at Honeywell Technology Laboratories under DARPA and Army AMCOM sponsorship.

(5) The AADL standard has been designed to be compatible with real-time operating system standards such as POSIX and ARINC 653.

(6) The AADL standard is aligned with Object Management Group (OMG) Unified Modeling Language (UML) and Modeling and Analysis of Real-Time Embedded systems (MARTE) through a standardized profile for AADL.

(7) The AADL standard includes a specification of an AADL-specific XML interchange format.

(8) The AADL standard provides guidelines for users to transition between AADL models and program source text written in Ada (ISO/IEC 8652/2007 (E) Ed.3) and C (ISO/IEC 9899:1999).

[1] This was the original name of the SAE AADL.

Introduction

(1) The SAE Architecture Analysis & Design Language (referred to in this document as AADL) is a textual and graphical language used to design and analyze the software and hardware architecture of performance-critical real-time systems. These are systems whose operation strongly depends on meeting non-functional system requirements such as reliability, availability, timing, responsiveness, throughput, safety, and security. AADL is used to describe the structure of such systems as an assembly of software components mapped onto an execution platform. It can be used to describe functional interfaces to components (such as data inputs and outputs) and performance-critical aspects of components (such as timing). AADL can also be used to describe how components interact, such as how data inputs and outputs are connected or how application software components are allocated to execution platform components. The language can also be used to describe the dynamic behavior of the runtime architecture by providing support to model operational modes and mode transitions. The language is designed to be extensible to accommodate analyses of the runtime architectures that the core language does not completely support. Extensions can take the form of new properties and analysis specific notations that can be associated with components and are standardized themselves.

(2) AADL was developed to meet the special needs of performance-critical real-time systems, including embedded real-time systems such as avionics, automotive electronics, or robotics systems. The language can describe important performance-critical aspects such as timing requirements, fault and error behaviors, time and space partitioning, and safety and certification properties. Such a description allows a system designer to perform analyses of the composed components and systems such as system schedulability, sizing analysis, and safety analysis. From these analyses, the designer can evaluate architectural tradeoffs and changes.

(3) Since AADL supports multiple and extensible analysis approaches, it provides the ability to analyze the cross cutting impacts of change in the architecture in one specification using a variety of analysis tools. AADL is designed to be used with analysis tools that support the automatic generation of the source code needed to integrate the system components and build a system executive. Since the models and the architecture specification drive the design and implementation, they can be maintained to permit model driven architecture based changes throughout the system lifecycle.

Information and Feedback

(1) The website at http://www.aadl.info is an information source regarding the SAE AADL standard. It makes available papers on AADL, its benefits, and its use. Also available are papers on MetaH, the technology that demonstrated the practicality of a model-based system engineering approach based on architecture description languages for embedded real-time systems.

(2) The website provides links to three SAE AADL related discussion forums:

- The SAE AADL User Forum to ask questions and share experiences about modeling with SAE AADL,
- The AADL Toolset User Forum to ask questions and share experiences with the Open Source AADL Tool Environment,(OSATE) and
- The SAE Standard Document Corrections & Improvements Forum that records errata, corrections, and improvements to the current release of the SAE AADL standard.

(3) The website provides information and a download site for the Open Source AADL Tool Environment. It also provides links to other resources regarding the AADL standard and its use.

(4) Questions and inquiries regarding working versions of annexes and future versions of the standard can be addressed to info@aadl.info.

(5) Informal comments on this standard may be sent via e-mail to errata@aadl.info. If appropriate, the defect correction procedure will be initiated. Comments should use the following format:

!topic Title summarizing comment

!reference AADL-ss.ss(pp)

!from Author Name yy-mm-dd

!keywords keywords related to topic

!discussion

text of discussion

(6) where ss.ss is the section, clause or subclause number, pp is the paragraph or line number where applicable, and yy-mm-dd is the date the comment was sent. The date is optional, as is the !keywords line.

(7) Multiple comments per e-mail message are acceptable. Please use a descriptive “Subject” in your e-mail message.

(8) When correcting typographical errors or making minor wording suggestions, please put the correction directly as the topic of the comment; use square brackets [ ] to indicate text to be omitted and curly braces { } to indicate text to be added, and provide enough context to make the nature of the suggestion self-evident or put additional information in the body of the comment, for example:

!topic [c]{C}haracter

!topic it[']s meaning is not defined

1 Scope

(1) This standard defines a language for describing both the software architecture and the execution platform architectures of performance-critical, embedded, real-time systems; the language is known as the SAE Architecture Analysis & Design Language (AADL). An AADL model describes a system as a hierarchy of components with their interfaces and their interconnections. Properties are associated to these constructions. AADL components fall into two major categories: those that represent the physical hardware and those representing the application software. The former is typified by processors, buses, memory, and devices, the latter by application software functions, data, threads, and processes. The model describes how these components interact and are integrated to form complete systems. It describes both functional interfaces and aspects critical for performance of individual components and assemblies of components. The changes to the runtime architecture are modeled as operational modes and mode transitions.

(2) The language is applicable to systems that are:

- real-time,
- resource-constrained,
- safety-critical systems,
- and those that may include specialized device hardware.

(3) This standard defines the core AADL that is designed to be extensible. While the core language provides a number of modeling concepts with precise semantics including the mapping to execution platforms and the specification of execution time behavior, it is not possible to foresee all possible architecture analyses. Extensions to accommodate new analyses and unique hardware attributes take the form of new properties and analysis specific notations that can be associated with components. Users or tool vendors may define these extensions. Extensions may be proposed as annex documents for inclusion in the AADL standard.

(4) This standard does not specify how the detailed design or implementation details of software and hardware components are to be specified. Those details can be specified by a variety of software programming and hardware description languages. The standard specifies relevant characteristics of the detailed design and implementation descriptions, such as source text written in a programming language or hardware description language, from an external (black box) perspective. These relevant characteristics are specified as AADL component properties, and as rules of conformance between the properties and the described components.

(5) This standard does not prescribe any particular system integration technologies, such as operating system or middleware application program interfaces or bus technologies or topologies. However, specific system architecture topologies, such as the ARINC 653 executives, can be modeled through software and execution platform components. AADL can be used to describe a variety of hardware architectures and software infrastructures. Integration technologies can be used to implement a specified system. The standard specifies rules of conformance between AADL system architecture specifications and actual system implementations.

(6) The standard was not designed around a particular set of tools. It is anticipated that systems and software tools will be provided to support the use of AADL.

1.1 Purpose/Extent

(1) The purpose of AADL is to provide a standard and sufficiently precise (machine-processable) way of modeling the architecture of an embedded, real-time system, such as an avionics system or automotive control system, to permit analysis of its properties, and to support the predictable integration of its implementation. Defining a standard way to describe system components, interfaces, and assemblies of components facilitates the exchange of engineering data between the multiple organizations and technical disciplines that are invariably involved in an embedded real-time system development effort. A precise and machine-processable way to describe conceptual and runtime architectures provides a framework for system modeling and analysis; facilitates the automation of code generation, system build, and other development activities; and significantly reduces design and implementation defects.

(2) AADL describes application software and execution platform components of a system, and the way in which components are assembled to form a complete system or subsystem. The language addresses the needs of system developers in that it can describe common functional (control and data flow) interfacing idioms as well as performance-critical aspects relating to timing, resource allocation, fault-tolerance, safety and certification.

(3) AADL describes functional interfaces and non-functional properties of application software and execution platform components. The language is not suited for detailed design or implementation of components. AADL may be used in conjunction with existing standard languages in these areas. AADL describes interfaces and properties of execution platform components including processor, memory, communication channels, and devices interfacing with the external environment. Detailed designs for such hardware components may be specified by associating source text written in a hardware description language such as VHDL[2]. AADL can describe interfaces and properties of application software components implemented in source text, such as threads, processes, and runtime configurations. Detailed designs and implementations of algorithms for such components may be specified by associating source text written in a software programming language such as Ada or C, or domain-specific modeling languages such as MatLab®/Simulink®[3].

(4) AADL describes how components are composed together and how they interact to form complete system architectures. Runtime semantics of these components are specified in this standard. Various mechanisms are available to exchange control and data between components, including message passing, event passing, synchronized access to shared components, and remote procedure calls. Thread scheduling protocols and timing requirements may be specified. Dynamic reconfiguration of the runtime architecture may be specified through operational modes and mode transitions. The language does not require the use of any specific hardware architecture or any specific runtime software infrastructure.

(5) Rules of conformance are specified between specifications written in AADL, source text and physical components described by those specifications, and physical systems constructed from those specifications. The AADL is not intended to describe all possible aspects of any possible component or system; selected syntactic and semantic requirements are imposed on components and systems. Many of the attributes of an AADL component are represented in an AADL model as properties of that component. The conformance rules of the language include the characteristics described by these properties as well as the syntactic and semantic requirements imposed on components and systems. Compliance between AADL specifications and items described by specifications is determined through analysis, e.g., by tools for source text processing and system integration.

(6) AADL can be used for multiple activities in multiple development phases, beginning with preliminary system design. The language can be used by multiple tools to automate various levels of modeling, analysis, implementation, integration, verification and certification.

1.2 Field of Application

(1) AADL was developed to model embedded systems that have challenging resource (size, weight, power) constraints and strict real-time response requirements. Such systems should tolerate faults and may utilize specialized hardware such as I/O devices. These systems are often certified to high levels of assurance. Intended fields of application include avionics systems, automotive systems, flight management systems, engine and power train control systems, medical devices, industrial process control equipment, robotics, and space applications. AADL may be extended to support other applications as the need arises.

1.3 Structure of Document

1.3.1 A Reader's Guide

(1) As necessary, the term AADL V2 will be used to refer to the revised version of AADL defined in this document.

---

[2] VHDL is the "Very-High-Speed-Integrated-Circuit Hardware Description Language. See IEEE VHDL Analysis and Standardization Group for details and status.

[3] MatLab and SimuLink are commercial tools available from The MathWorks.

(2) The AADL standard consists of this core language document and a set of annex documents of standardized extensions. This core language document contains a number of sections and appendices. The sections define the core AADL. The appendices provide additional information, both normative and informative about the core language. Annex documents introduce additional standardized properties and possibly language extensions in the form of specialized notations.

(3) AADL concepts are introduced in section 3, Architecture Analysis & Design Language Summary. They are defined with full syntactic and semantic descriptions as well as naming and legality rules in succeeding sections. The vocabulary and symbols of AADL are defined in Section 15. Appendix B , Glossary, provides informative definitions of terms used in this document. Other appendices include a Syntax Summary and Predeclared Property Sets. The remainder of this section introduces notations used in this document and discusses standard conformance.

(4) This core AADL document consists of the following:

- Section 2, References, provides normative and applicable references as well as terms and definitions.
- Section 3, Architecture Analysis & Design Language Summary, introduces and defines the concepts of the language.
- Section 4, Components, Packages, and Annexes, defines the common aspects of components, which are the design elements of AADL, as well as component template parameterization. It also introduces the package, which allows organization of the design elements in the design space. This section closes with a description of annex subclauses and libraries as annex-specific notational extensions to the core AADL.

(5) The next sections introduce the language elements for modeling application and execution platform components in modeled systems or systems of systems.

- Section 5, Software Components, defines those modeling elements of AADL that represent application system software components, i.e., data, subprogram, subprogram group, thread, thread group, and process.
- Section 6, Execution Platform Components, defines those modeling elements of AADL that model execution platform components, i.e., processor, virtual processor, memory, bus, virtual bus, and device.
- Section 7, System Composition, defines system as a compositional modeling element that combines execution platform and application system software components.
- Section 8, Features and Shared Access, defines the features of components that are connection points with other components. These consist of ports, subprogram parameters, provided and required access to data, subprograms, and buses, as well as grouping of features into feature groups.
- Section 9, Connections , defines the constructs to express interaction between components in terms of connections between component features.
- Section 10, Flows, defines the constructs to express flows through a sequence of components, and connections.
- Section 11, Properties, defines the AADL concept of properties including property sets, property value association, property type, and property declaration. Property associations and property expressions are used to specify values. Property set, property type, and property name declarations are used to extend AADL with new properties.
- Section 12, Modes, defines modes and mode transitions to support modeling of operational modes with mode-specific system configurations and property values.
- Section 13, Operational System, defines the concepts of system instance and binding of application software to execution platforms. This section defines the execution semantics of the operational system including the semantics of system-wide mode switches.
- Section 14, Layered System Architectures, defines support for modeling layered architectures.

(6) Section 15, Lexical Elements, defines the basic vocabulary of the language. As defined in this section, identifiers in AADL are case insensitive. Identifiers differing only in the use of corresponding upper and lower case letters are considered as the same. Similarly, reserved words in AADL are case insensitive.

(7) The following Appendix sections complete the definition of the core AADL.

- Appendix A Predeclared Property Sets, contains the standard AADL set of predeclared properties, property types, and property constants.
- Appendix B Glossary, contains a glossary of terms.
- Appendix C Syntax Summary, contains a summary of the syntax as defined in the sections of this document.
- Appendix D Graphical AADL Notation, defines a graphical representation of AADL.

- Appendix E AADL Meta Model and XML Specification, defines an XML-based interchange format in form of an XMI meta model and an XML schema.
- Appendix F Unified Modeling Language (UML) Profile, defines a profile for UML that extends and tailors UML to support modeling in terms of AADL concepts. This profile is defined in the context of the Object Management Group (OMG) Modeling and Analysis of Real-Time Embedded systems (MARTE).
- Appendix G Profiles and Extensions, contains profiles and extensions that have been approved by the standards body.

(8) The annex documents introduce additions and extensions to the core AADL.

- Annex Document A Code Generation, provides guidance for automatic generation and integration of runtime systems and application code in different implementation languages. It defines a standardized set of properties for recording mappings from the AADL model to source text and for automatic code generation.
- Annex Document B Data Modeling, provides guidance on data modeling and how to map relevant data modeling information into an AADL model if desirable. It defines a standardized set of properties and basic data types in support of data modeling.
- Annex Document C Error Model, defines a standardized core language extension in the form of a sublanguage notation and properties the component to support annotating AADL models with safety-criticality and dependability related information of a system.
- Annex Document D Behavior Model, defines a standardized core language extension in the form of a sublanguage notation to specify the behavior of AADL components as AADL model annotations.

(9) The core language and the annex documents are *normative*, except that the material in each of the items listed below is informative:

- Text under a NOTES or Examples heading.
- Each clause or subclause whose title starts with the word “Example" or “Examples".

(10) All implementations shall conform to the core language. In addition, an implementation may conform separately to one or more Annexes that represent extensions to the core language.

(11) The following appendices and annexes are informative and do not form a part of the formal specification of AADL:

- Appendix B Glossary
- Appendix C Syntax Summary
- Appendix G Profiles and Extensions

1.3.2 Structure of Clauses and Subclauses

(1) Each section of the core standard is divided into clauses and subclauses that have a common structure. Each section, clause, and subclause first introduces its subject and then presents the remaining text in the following format. Not all headings are required in a particular clause or subclause. Headings will be centered and formatted as shown below.

(2) All paragraphs are numbered with numbering restarting with each section. Naming rules, legality rules, and consistency rules have their own paragraph numbering also restarting with each section. They can be identified by section number and paragraph number.

Syntax

(3) Syntax rules, concerned with the organization of the symbols in the AADL expressions, are given in a variant of Backus-Naur-Form (BNF) that is described in detail in Section 1.5.

Naming Rules

(4) *Naming rules* define rules for names that represent defining identifiers and references to previously defined identifiers. Each rule is labeled by (N#), where # is a natural number restarting with 1 for each section.

Legality Rules

(5) *Legality rules* define semantic restrictions on AADL specifications. Legality rules must be validated by AADL processing tools when a model is loaded into the tool. Each rule is labeled by (L#), where # is a natural number restarting with 1 for each section.

Consistency Rules

(6) *Consistency rules* define consistency restrictions on system instances. A consistency rule must be validated by AADL processing tools upon a user request or when an analysis method that relies on the rule is invoked. Each rule is labeled by (C#), where # is a natural number restarting with 1 for each section.

Standard Properties

(7) *Standard properties* define the properties that are defined within this standard for various categories of components. The listed properties are fully described in Appendix A .

Semantics

(8) *Semantics* describes the static and dynamic meanings of different AADL constructs with respect to the system they model. The semantics are concerned with the effects of the execution of the constructs, not how they would be specifically executed in a computational tool.

Runtime Support

(9) AADL concepts may require runtime support through an operating system or other runtime system on a processor. Such service calls may be programmed explicitly in the application source code, or may be part of a runtime system generated from an AADL specification. The Code Generation Annex provides guidance on the use of these runtime services by application code or the runtime system.

Processing Requirements and Permissions

(10) AADL specifications may be processed manually or by tools for analysis and generation. This section documents additional requirements and permissions for determining compliance. Providers of processing method implementations must document a list of those capabilities they support and those they do not support.

NOTES:

Notes emphasize consequences of the rules described in the (sub)clause or elsewhere. This material is informative.

Examples

(11) Examples illustrate the possible forms of the constructs described. This material is informative.

1.4 Error, Exception, Anomaly and Compliance

(1) AADL can be used to specify dependable systems. A system can be compliant with its specification and this standard even when that system contains failed components that no longer satisfy their specifications. This section defines the terms fault, error, exception, anomaly and noncompliance [IFIP WG10.4-1992]; and defines how those terms apply to AADL specifications, physical components (implementations), models of components, and tools that accept AADL specifications as inputs.

(2) A *fault* is defined to be an anomalous undesired change in execution behavior, possibly resulting from an anomalous undesired change in data being accessed by a thread or from violation of a compute time or deadline constraint. A fault in a physical component is a root cause that may eventually lead to a component error or failure. A fault is often a specific event such as a transistor burning out or a programmer making a coding mistake.

(3) An *error* in a physical component occurs when an existing fault causes the internal state of the component to deviate from its nominal or desired operation. For example, a component error may occur when an add instruction produces an incorrect result because a transistor in the adding circuitry is faulty.

(4) A *failure* in a physical component occurs when an error manifests itself at the component interface. A component fails when it does not perform its nominal function for the other parts of the system that depend on that component for their nominal operation.

(5) A component failure may be a fault within a system that contains that component. Thus, the sequence of fault, error, failure may repeat itself within a hierarchically structured system. *Error propagation* occurs when a failed component causes the containing system or another dependent component to become erroneous.

(6) A component may persist in a faulty state for some period of time before an error occurs. This is called *fault latency*. A component may persist in an erroneous state for some period of time before a failure occurs. This is called *error latency*.

(7) An *exception* represents a kind of exceptional situation; it may occur for an erroneous or failed component when that error or failure is detected, either by the component itself or another component with which it interfaces. For example, a fault in a software component that eventually results in a divide-by-zero may be detected by the processor component on which it depends. An exception is always associated with a specific component. This document defines a standard model for exceptions for certain kinds of components (e.g., defines standard recovery sequences and standard exception events).

(8) An *anomaly* occurs when a component is in an erroneous or failed state that does not result in a standard exception. Undetected errors may occur in systems. A detected error may be handled using mechanisms other than the standard exception mechanisms. For example, an error may propagate to multiple components before it is detected and mitigated. This standard defines nominal and exceptional behaviors for components. Anomalies are any other undefined erroneous component behaviors, which are nevertheless considered compliant with this standard.

(9) An AADL specification is *compliant* with the AADL core language standard if it satisfies all the syntactic and legality rules defined in Sections 4 - 15. An AADL specification is compliant with an AADL Annex standard if it satisfies all the syntactic and legality rules defined in the respective normative Annex.

(10) A component or system is *compliant* with an AADL specification of that component or system if the nominal and exceptional behaviors of that component or system satisfy the applicable semantics of the AADL specification, as defined by the semantic rules in this standard. A component or system may be a physical implementation (e.g., a piece of hardware), or may be a model (e.g., a simulation or analytic model). A model component or system may exhibit only partial semantics (e.g., a schedulability model only exhibits temporal semantics). Physical components and systems must exhibit all specified semantics, except as permitted by this standard.

(11) *Noncompliance* of a component with its specification is a kind of design fault. This may be handled by run-time fault-tolerance in an implemented actual system. A developer is permitted to classify such components as anomalous rather than noncompliant.

(12) A tool that operates on AADL specifications is *compliant* with the core language standard if the tool checks for compliance of input specifications with the syntactic and legality rules defined herein, except where explicit permission is given to omit a check; and if all physical or model components or systems generated by the tool are compliant with the specifications used to generate those components or systems. The AADL standard allows profiles of language subsets to be defined and requires a minimum subset of the language to be supported (see Appendix G). A tool must clearly specify any portion of the language not supported and warn the user if a specification contains unsupported language constructs, when appropriate. A tool is compliant with the XMI interchange format if it supports saving and reading of AADL model in the XMI interchange format. A tool is compliant with a language extension annex if the tool checks for compliance of input specifications with the syntactic and legality rules defined in the respective annex document.

(13) Compliance of an AADL specification with the syntactic and legality rules can be automatically checked, with the exception of a few legality rules that are not in general tractably checkable for all specifications. Compliance of a component or system with its specification, and compliance of a tool with this standard, cannot in general be fully automatically checked. A verification process that assures compliance to the degree required for a particular purpose must be used to perform the latter two kinds of compliance checking.

1.5 Method of Description and Syntax Notation

(1) The language is described by means of a context-free syntax together with context-dependent requirements expressed by narrative rules. The meaning of a construct in the language is defined by means of narrative rules.

(2) The context-free syntax of the language is described using the variant Backus-Naur Form (BNF) [BNF 1960] as defined herein.

- Lower case words in `courier new` font, some containing embedded underlines, are used to denote syntactic categories. A syntactic category is a nonterminal in the grammar. For example:
  `component_feature_list`

- Boldface words are used to denote reserved words, for example:
  `implementation`

- A vertical line separates alternative items.
  `software_category ::= thread | process`

- Square brackets enclose optional items. Thus the two following rules are equivalent.
  `property_association ::= property_name => [ constant ] expression`

  ```
  property_association ::=

       property_name => expression

     | property_name => constant expression
  ```

- Curly brackets with a * symbol enclose a repeated item. The item may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule. Thus the two following rules are equivalent.
  `declaration_list ::= declaration { declaration }*`

  ```
  declaration_list ::= declaration

                     | declaration declaration_list
  ```

- Curly brackets with a + symbol specify a repeated item with one or more occurrences. Thus the two following rules are equivalent.
  `declaration_list ::= { declaration }+`

  `declaration_list ::= declaration { declaration }*`

- Parentheses (round brackets) enclose several items to group terms. This capability reduces the number of extra rules to be introduced. Thus, the first rule is equivalent with the latter two.

```
property_association ::= identifier ( =>| +=>) property_expression

property_association ::= identifier assign property_expression

assign ::= => | +=>
```

- Square brackets, curly brackets, and parentheses may appear as delimiters in the language as well as meta-characters in the grammar. Square, curly, and parentheses that are delimiters in the language will be written in bold face in grammar rules, for example:

```
property_association_list ::=

    { property_association { ; property_association }* }
```

- The syntax rules may preface the name of a nonterminal with an italicized name to add semantic information. These italicized prefaces are to be treated as comments and not a part of the grammar definition. Thus the two following rules are equivalent.

```
component ::= identifier : component_classifier ;

component ::= component_identifier : component_classifier ;
```

- A construct is a piece of text (explicit or implicit) that is an instance of a syntactic category, for example:

```
My_GPS: thread GPS.dualmode ;
```

(3) The syntax description has been developed with an emphasis on an abstract syntax representation to provide clarity to the reader.

1.6 Method of Description for Discrete and Temporal Semantics

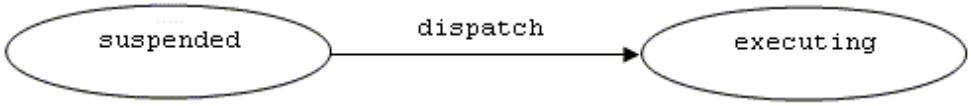
(1) Discrete and temporal semantics of the language are defined in sections that define AADL concepts using a concurrent hierarchical hybrid automata notation, together with additional narrative rules about those diagrams. This notation consists of a hierarchical finite state machine notation, augmented with real-valued variables to denote time and time-varying values, and with edge guard and state invariant predicates over those variables to define temporal constraints on when discrete state transitions may occur.

(2) A semantic diagram defines the nominal scheduling and reconfiguration behavior for a modeled system as well as scheduling and reconfiguration behavior when failures are detected. A physical realization of a specification may violate this definition, for example due to runtime errors. A violation of the defined semantics is called an anomalous behavior. Certain kinds of anomalous behaviors are permitted by this standard. Legal anomalous behaviors are defined in the narrative rules.

(3) Semantics for individual components are defined using a sequential hierarchical hybrid automaton. System semantics are defined as the concurrent composition of the hybrid automata of the system components.

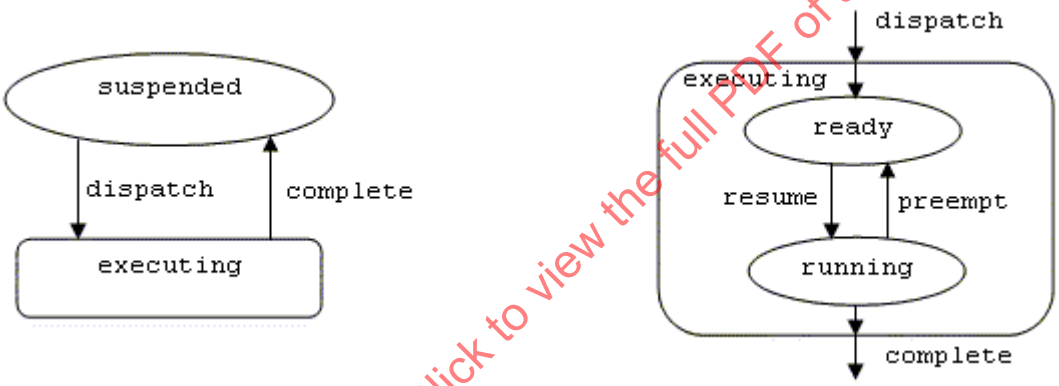
(4) Ovals labeled with lower case phrases are used to denote discrete states. A component may remain in one of its discrete states for an interval of time whose duration may be zero or greater. Every semantic automaton for a component has a unique initial discrete state, indicated by a heavy border. For example,

(5) Directed edges labeled with one or more comma-separated, lower case phrases are used to denote possible transitions between the discrete states of a component. Transitions over an edge are logically instantaneous, i.e., the time interval in which a transition from a discrete state (called the source discrete state) to a discrete state (called the destination discrete state) has duration 0. For example,

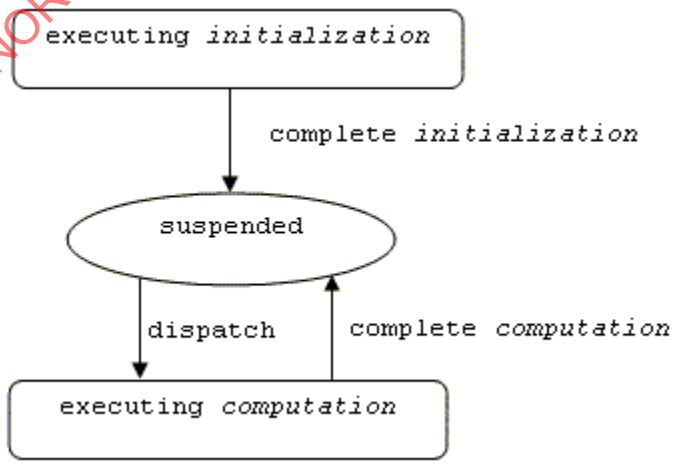


(6) Permissions that allow a runtime implementation of a transition to occur over an interval of time are expressed as narrative rules. However, all implemented transitions must be atomic with respect to each other, all observable serializations must be admitted by the logical semantics, and all temporal predicates as defined in subsequent paragraphs must be satisfied.

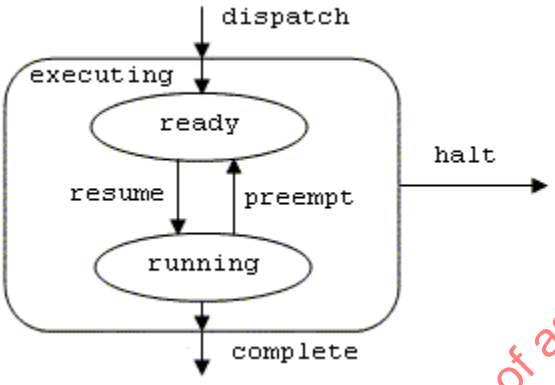
(7) Hybrid automaton can have hierarchical states. Oblong boxes labeled with lower case phrases denote abstract discrete states, for which another hybrid semantics diagram with an identically labeled oblong box for which another hybrid semantics diagram with an identically labeled oblong box specifies the discrete states and edges that make up that abstract discrete state. For example,



(8) Abstract discrete states are reusable, i.e., a hybrid semantics diagram can contain several oblong boxes with the same label An abstract state label or an edge label may include italicized letters that are not a part of the formal name but are used to distinguish multiple instances. For example, both abstract discrete states below will be defined by a single diagram labeled `executing`.



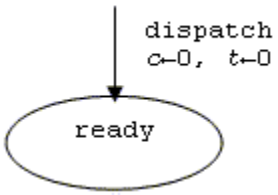
(9) If there is an external edge that enters or exits an abstract discrete state in the defining diagram for, and there are no edges within that definition that connect any internal discrete state with that external edge, then there implicitly exist edges from every contained discrete state in the defining diagram to or from that external edge. In that case, a transition into or out of an abstract discrete state represents transitions into or out any of its internal states. For example, in the following diagram there is an implicitly defined `halt` edge out of both the `ready` and the `running` discrete states.



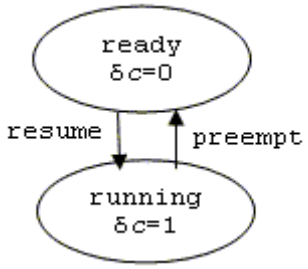
(10) Real-valued variables whose values are time-varying may appear in expressions that annotate discrete states and edges of hybrid semantic diagrams. Specific forms of annotation are defined in subsequent paragraphs. The set of real-valued variables associated with a semantic diagram are those that appear in any expression in that diagram, or in any of the defining diagrams for abstract discrete states that appear in that diagram. Real-valued time-varying variables will be named using an italicized front. The initial values for the real-valued time-varying variables of a hybrid semantic diagram are undefined whenever they are not explicitly defined in narrative rules.

(11) In addition to standard rational literals and arithmetic operators, expressions may also contain functions of discrete variables. The names of functions and discrete variables will begin with upper case letters. The semantics for function symbols and discrete variables will be defined using narrative rules. For example, the subexpression `Max(Compute_Time)` may appear in a semantic diagram, together with a narrative rule stating that the value is the maximum value of a range-valued component property named `Compute_Time`.

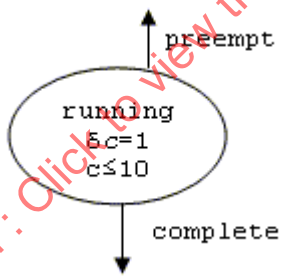
(12) Edges may be annotated with assignments of values to variables associated with the semantic diagram. When a transition occurs over an edge, the values of the variables are set to the assigned values. For example, in the following diagram, the values of the variables *c* and *t* are set to 0 when the component transitions into the `ready` discrete state.



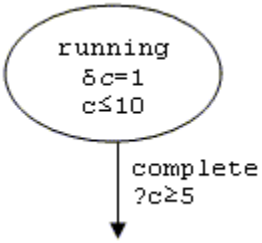
(13) Discrete states may be annotated with expressions that define the possible rates of change for real-valued variables during the duration of time a component is in that discrete state. The rate of a variable is denoted using the symbol δ, for example δ*x*=[0,1] (the rate of the variable *x* may be any real value in the range of 0 to 1). If, rates of change are not explicitly shown within a discrete state for a time-varying variable, then the rate of change of that variable in that state is defined to be 1. For example, in the following diagram the rate of change for the variable *c* is 1 while the component is in the discrete state `running`, but its value remains fixed while the component is in the `ready` state, equal to the value that existed when the component transitioned into the `ready` state.



(14) A discrete state may be annotated with Boolean-valued expressions called invariants of that discrete state. In this standard, all semantic diagrams are defined so that the values of the variables will always satisfy the invariants of a discrete state for every possible transition into that discrete state. A transition must occur out of a discrete state before the values of any time-varying variables cause any invariant of that discrete state to become false. Invariants are used to define bounds on the duration of time that a component can remain in a discrete state. For example, in the following diagram the component must transition out of the `running` state before the value of the variable *c* exceeds 10.

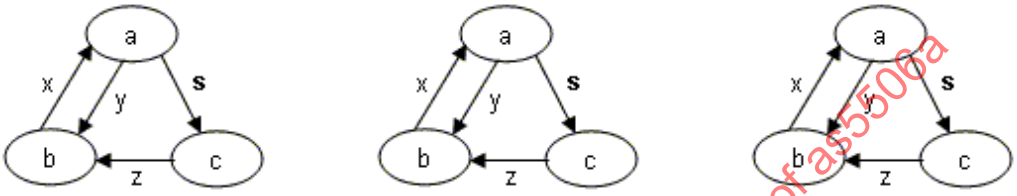


(15) An edge may be annotated with Boolean-valued expressions called guards of that edge. A transition may occur from a source discrete state to a destination discrete state only when the values of the variables satisfy all guards for an edge between those discrete states. A guard on an edge is evaluated before any assignments on that edge are performed. For example, in the following diagram the component may only `complete` when the value of the variable *c* is 5 or greater (but must `complete` before *c* exceeds 10 because of the invariant).



(16) A sequential semantic automaton defines semantics for a single component. A system may contain multiple components. The semantics of a system are defined to be the concurrent composition of the sequential semantic automata for each component. Except as described below, every component is represented by a copy of its defined semantic automaton. All discrete states and labels, all edges and labels, and all variables, are local to a component. The set of discrete states of the system is the cross-product of the sets of discrete states for each of its cross product components. The set of transitions that may occur for a system at any point in time is the union of the transitions that may occur at that instant for any of its components.

(17) If an edge label appears in boldface, then a transition may occur over that edge only when a transition occurs over all edges having that same boldface label within the synchronization scope for that label. The synchronization scope for a boldface label is indicated in parentheses. For example, if a transition occurs over an edge having a boldface label with a synchronization scope of process, then every thread contained in that process in which that boldface label appears anywhere in its hybrid semantic diagram must transition over some edge having that label. That is, transitions over edges with boldface labels occur synchronously with all similarly labeled edge transitions in all components associated with the component with the specified synchronization scope as described in the narrative. Furthermore, every component in that synchronization scope that might participate in such a transition in any of its discrete states must be in one of those discrete states and participate in that transition. For example, when the synchronization scope for the edge label **s** is the same for all three of the following concurrent semantic automata, a transition over the edge labeled **s** may only occur when all three components are in their discrete states labeled `a`, and all three components simultaneously transition to their discrete states labeled `c`.



(18) If a variable appears in boldface, then there is a single instance of that variable that is shared by all components in the synchronization scope of the variable. The synchronization scope for a boldface variable will be defined in narrative rules.

2 References

2.1 Normative References

(1) The following normative documents contain provisions that, through reference in this text, constitute provisions of this standard.

(2) IEEE/ANSI 610.12-1990 [IEEE/ANSI 610.12-1990], IEEE Standard Glossary of Software Engineering Terminology.

(3) ISO/IEC 9945-1:1996 [IEEE/ANSI Std 1003.1, 1996 Edition], Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) [C Language].

(4) ISO/IEC 14519:1999 [IEEE/ANSI Std 1003.5b-1999], Information Technology – POSIX Ada Language Interfaces – Binding for System Application Program Interface (API) – Real-time Extensions.

(5) ISO/IEC 8652:2007 Ed.3, Information Technology – Programming Languages – Ada Reference Manual.

(6) ISO/IEC 9899:1999, Information Technology – Programming Languages – C.

(7) Unified Modeling Language Specification [UML 2007, version 2.1.1], August 2007, version 2.1.1.

(8) SAE AS5506, Architecture Analysis & Design Language (AADL), November 2004.

(9) SAE AS5506/1, Architecture Analysis & Design Language (AADL) Annex Volume 1, June 2006.

2.2 Informative References

(1) The following informative references contain background information about the items with the citation.

(2) [BNF 1960] NAUR, Peter (ed.), "Revised Report on the Algorithmic Language ALGOL 60," *Communications of the ACM*, Vol. 3 No. 5, pp. 299-314, May 1960.

(3) [IFIP WG10.4-1992] IFIP WG10.4 on Dependable Computing and Fault Tolerance, 1992, J.-C. Laprie, editor, “Dependability: Basic Concepts and Terminology,” *Dependable Computing and Fault Tolerance*, volume 5, Springer-Verlag, Wien, New York, 1992.

(4) [Henz 96] “Theory of Hybrid Automata”, Thomas A. Henzinger, Electrical Engineering and Computer Science, University of California at Berkley, *Proceedings of the 11th Annual Symposium on Logic in Computer Science* (LICS), IEEE Computer Society Press, 1996, pp. 278-292

2.3 Terms and Definitions

(1) Terms are introduced throughout this standard, indicated by *italic* type. Informational definitions of terms are given in Appendix B , Glossary. Definitions of terms used from other standards, such as the IEEE *Standard Glossary of Software Engineering Terminology* [IEEE Std. 610.12-1990], ISO/IEC 9945-1:1996 [IEEE/ANSI Std 1003.1, 1996 Edition], *Information Technology – Portable Operating System Interface (POSIX)*, or IFIP WG10.4 *Dependability: Basic Concepts and Terminology* [IFIP WG10.4-1992], are so marked. Terms not defined in this standard are to be interpreted according to the Webster's Third New International Dictionary of the English Language. Terms explicitly defined in this standard are not to be presumed to refer implicitly to similar terms defined elsewhere. A full description of the syntax and semantics of the concept represented by the terms is found in the respective document sections, clauses, and subclauses.

3 Architecture Analysis & Design Language Summary

(1) This section provides an informative overview of AADL concepts, structure, and use. In this section the first appearance of a term that has a specific meaning in this standard will be italicized.

(2) An AADL specification represent s a component model of a computer system runtime architecture that consists of the *application software (typically embedded, safety-critical, mission-critical, or performance-critical)*, and the *execution platform*, i.e., the computing hardware and the physical system. A *component* represents a part of a system and interacts with other components. A *system* is hierarchically composed of interacting components. An AADL specification consists of *package* declarations and *property set* declarations. AADL packages contain component specifications, i.e., AADL *component types* and *component implementations*, as well as *feature group types*, and *annex libraries*. This standard defines *data*, *subprogram*, *subprogram group*, *thread*, *thread group*, and *process* as application software component categories, *memory*, *bus*, *virtual bus*, *processor*, *virtual processor*, and *device* as execution platform component categories, and *system* and *abstract* as general compositional components. *Abstract* is a generic component category that can be refined into any of the other categories. They form the core of the AADL modeling vocabulary.

(3) A *component type* specifies an external interface that other components can operate against in terms of *features*, i.e., interaction points with other components, *flow specifications* from component inputs to component outputs, *modes* as operational states, and *properties* to characterize a component. Implementations of the component are required to satisfy this specification.

(4) A *feature* describes an interface of a component through which control and data may be provided to or required from other components. Features can be *ports* to support directional flow of control and data, *subprograms* to represent procedure calls, and shared *access* to data, subprograms, subprogram groups, and bus components. Features can be grouped together into *feature groups*.

(5) A *component implementation* specifies a realization of the component in terms of *subcomponents*, *connections* between the features of those subcomponents, *flows* across a sequence of subcomponents, modes, and properties. AADL allows multiple component implementations to associated with a component type to represent component variants.

(6) A *subcomponent* declares a component instance that is contained in another component by naming its *component classifier*, i.e., a component type or component implementation. The component hierarchy of a system instance is determined by recursively instantiating the subcomponents of a top-level system.

(7) Any namable model element, e.g., components, features, modes, connections, flows, and subprogram calls, can have properties. Properties are used to represent attributes and other characteristics, such as the period and deadline of threads. When properties are associated with declarations of component types, component implementations, features, subcomponents, connections, flows, and modes, they apply to all respective instances within a system instance. The AADL also supports the specification of instance specific values of any unit in the containment hierarchy of a system instance. AADL tools may record these values for use in the analysis of the system instance or for use in the construction of new system instances.

(8) *Property sets* are used to define *properties*, *property constants*, and *property types*. This standard defines a set of predeclared properties and property types. Additional properties and property types to support new forms of system analysis can be introduced through property sets. For example, a predeclared property is used to specify the period of a thread. An example of a user-defined property in a property set is a security level property.

(9) AADL packages provide a library-like structure for organizing component classifiers, and feature group types, and annex libraries into separate namespaces, similar to Java packages that are used to organize Java class declarations. Packages can have a nested naming hierarchy, but this hierarchy does not impose any restrictions on whether a package is accessible by other packages. A component classifier in a package is referenced by qualifying its name with the package name. Only those classifiers that have been placed in the *public* section of a package are accessible to other packages. Furthermore, the packages being named in the reference must be listed in the *with* clause of the referencing package, i.e., a *with* clause limits the use of other packages in a given package.

(10) AADL support the specification of partial models such as models of the application software only, the hardware only, specification of a top-level architecture in terms of its subsystems without their realizations, and specification of component templates. Component templates are incompletely specified component classifiers that may be parameterized by component classifier, feature, and feature group type *prototypes*. These component templates may be later refined through component classifier *extension* declarations. Component classifier extensions can complete the component specification by supplying an actual for a prototype and by completing partial declaration through *refinement*. Component extension declarations can also specify new component classifiers by adding features, subcomponents, connections, flows, and properties. This allows conceptual and reference architectures to be specified and to be refined into fully specified runtime architectures, and partial system specifications to evolve into fully specified and configured systems including the deployment of application software on the computing hardware

(11) Application software components model *source text, virtual address spaces, concurrent tasks* and their interactions. *Source text* can be written in a programming language such as Ada, C, or Java, or domain-specific modeling languages such as Simulink, SDL, ESTEREL, LUSTRE, and UML, for which executable code may be generated. The source text modeled by a software component may represent a partial application program or model (e.g., they form one or more independent compilation units as defined by the applicable programming language standard). Rules and permissions governing the mapping between AADL specification and source text depend on the applicable programming or modeling language standard. Predeclared component properties identify the source text container and the mapping of AADL concepts to source text declarations and statements. These properties also specify memory and execution times requirements and other known characteristics of the component.

(12) AADL *data* components represent static data in source text. These data components can be accessed by one or more threads and processes; they do so by indicating that they require access to the external data component. Concurrent access to data is managed by the appropriate concurrency control protocol as specified by a property. Realizations of such protocols can be documented through an appropriate annex declaration, for example, expressed in a Behavior Annex subclause (see Annex Document D).

(13) Data types in the source text are modeled by the declarations: data component type and data component implementation. Thus, a data component classifier represents the data type of data components, ports, and subprogram parameters. The Data Modeling Annex (see Annex Document B) provides guidance on how to approach data modeling with AADL.

(14) The *subprogram* component models source text that is executed sequentially. Subprograms are callable from within threads and subprograms. Subprograms may require access to data components and may contain data subcomponents to represent local variables. Subprogram groups represent source code libraries.

(15) AADL *thread* components model concurrent tasks or active objects, i.e., concurrent logical threads. Each logical thread represents an execution sequence through source text (or more exactly, through binary images produced from the compilation, linking and loading of source text). A scheduler manages the execution of a thread. Logical threads may be executed by separate operating system (OS) threads, or they may be combined into a single operating system thread. The dynamic semantics for a thread are defined in this standard using hybrid automata. The threads can be in states such as suspended, ready, and running. State transitions occur as a result of dispatch requests, faults, and runtime service calls. They can also occur if time constraints are exceeded. Error detection and recovery semantics are specified. Dispatch semantics are given for standard dispatch protocols such as periodic, sporadic, aperiodic, timed, hybrid, and background threads. Additional dispatch protocols may be defined. Threads can contain subprogram and data components, and provide or require access to data components.

(16) AADL *thread groups* support structural grouping of threads within a process. A thread group may contain data, thread, and thread group subcomponents. A thread group may require and provide access to data components.

(17) AADL *process* components model space partitions in terms of virtual address spaces containing source text that forms complete programs as defined in the applicable programming language standard. Access protection of the virtual address space is by default enforced at runtime, but can be disabled if specified by the property `Runtime_Protection`. The binary image produced by compiling and linking this source text must execute properly when loaded into a unique virtual address space. As processes do not represent concurrent tasks, they must contain at least one thread. Processes can contain thread groups, threads, and data components, and can access or share data components.

(18) Execution platform components represent computing hardware components that are capable of scheduling threads, of enforcing specified address space protection at runtime, of storing source text code and data and of performing communication for application system connections. The device component represents elements of the physical environment that an embedded system interacts with, such as sensors, actuators, or engines.

(19) AADL *processor* components are an abstraction of hardware and software that is responsible for scheduling and executing threads. In other words, a processor may include functionality provided by operating systems. Alternatively, operating systems can be modeled like application components. Processors can contain memory and require access to buses. Processors can support different scheduling protocols. Threads are bound to processors for scheduling and execution.

(20) AADL *virtual processors* represent virtual machines or hierarchical schedulers. Threads can be bound to them. Virtual processors can be used in two ways. Processors and virtual processors can be subdivided into virtual processors by declaring virtual processor subcomponents. Virtual processors can also be declared separately and explicitly bound to processors and virtual processors.

(21) AADL *memory* components model randomly accessible physical storage such as RAM or ROM. Memories have properties such as the number and size of addressable storage locations. Binary images of source text are bound to memory. Memory can contain nested memory components. Memory components require access to buses.

(22) AADL *bus* components model communication channels that can exchange control and data between processors, memories, and devices. A bus is typically hardware that supports specific communication protocols, possibly implemented through software. Processors, memories, and devices communicate by accessing a shared bus. Buses can be directly connected to other buses. Logical connections between threads that are bound to different processors transmit their information across buses that provide the physical connection between the processors. Buses can require access to other buses.

(23) AADL *virtual bus* components represent virtual channels or communication protocols that perform transmission within processors or across buses. Virtual buses can be subcomponents of buses and virtual buses, or virtual buses can be declared separately and explicitly bound to buses and virtual buses.

(24) AADL *device* components model physical entities in the external environment, e.g., a GPS system, or entities that interface with an external environment, e.g., sensors and actuators as interface between a physical plant and a control system. Devices may represent a physical entity of the modeled system or its (simulated) software equivalent. Examples of devices are timers, which exhibit simple behavior, or a camera or GPS, which exhibit complex behavior. Devices are logically connected to application software components and physically connected to processors via buses. They cannot store nor execute external application software source text themselves, but may include driver software executed on a connected processor. A device requires access to buses.

(25) AADL *system* components model hierarchical compositions of software and execution platform components. A system may directly contain data, subprogram, subprogram groups, process, memory, processor, virtual processor, bus, virtual bus, device, system as well as abstract subcomponents. Thread and thread group subcomponents must be declared in processes and are indirectly part of a system that contains these processes. A system component may require and provide access to data and bus components. Execution platform component can be system components in their own right and be modeled using system implementations. For example, a system implementation can be associated with a device that models a camera. This system implementation describes the internal of the camera in terms of the CCD sensor a device, a DSP processor, a general purpose processor as well a software that implements the image processing and download capability of the camera.

(26) AADL *modes* represent the operational states of software, execution platform, and compositional components in the modeled physical system. A component can have mode-specific property values. A component can also have mode-specific configurations of different subsets of subcomponents and connections. In other words, a mode change can change the set of active components and connections. Mode transitions model dynamic operational behavior that represents switching between configurations and changes in component-internal characteristics, such as conditional execution source text sequences or operational states of a device, that are reflected in property values. Other examples of mode-specific property values include the period or the worst-case execution time of a thread. A change in operating mode can have the effect of activating and deactivating threads for execution and changing the pattern of connections between threads. A mode subclause in a component implementation specifies the mode states and mode change behavior in terms of transitions; it specifies the events as transition triggers. Subcomponent and connection declarations as well as property associations declare their applicability (participation) in specific modes.

(27) This standard defines several categories of features: *data port*, *event port*, *event data port*, *feature group*, *subprogram parameter*, and *provided* and *required access* to data, subprograms, and buses. Data ports represent connection points for transfer of state data such as sensor data. Event ports represent connection points for transfer of control through raised events that can trigger thread dispatch or mode transition. Event data ports represent connection points for transfer of events with data, i.e., messages that may be queued. Feature groups support grouping of ports and other features, such that they can be connected to other components through a single connection. Provided subprogram access features represent entrypoints to code sequences in source text that is associated with a data type or a thread that can be called locally or remotely. Subprogram parameters represent in and out parameters of a subprogram. Data component access represents provided and required access to shared data. Bus component access represents provided and required access to buses for processors, memory, and devices.

(28) AADL *connections* specify interaction between components at runtime. A semantic connection is represented by a set of one or more connection declarations that follow the component hierarchy from the ultimate connection source to the ultimate connection destination. For example, in Figure 1 there is a connection declaration from a thread out port in Thread1 to a containing process out port in Process3. This connection is continued with a connection declaration within System1 from Process3's out port to Process4's in port. The connection declaration continues within Process4 to the thread in port contained in Thread2. Collectively, this sequence of connections defines a single semantic connection between Thread1 and Thread2. Threads, processes, systems, and ports are shown in graphical AADL notation. For a full description of the graphical AADL notation see Appendix D .

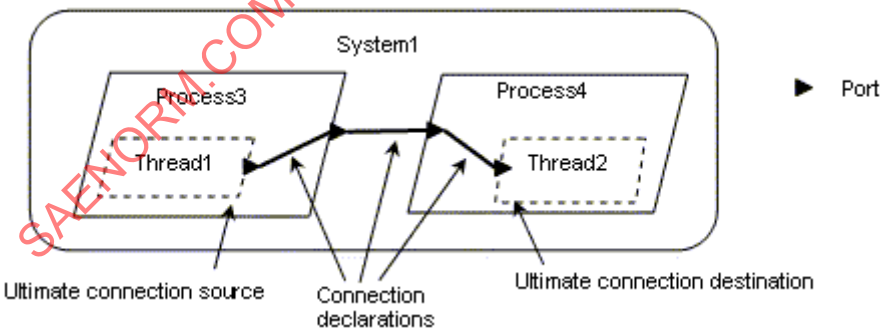


Figure 1 Example Semantic Connections

(29) *Flow specifications* describe externally observable flow of information in terms of application logic through a component. Such logical flows may be realized through ports and connections of different data types and a combination of data, event, and event data ports, as well as through data components. Flow specifications represent *flow sources*, i.e., flows originating from within a component, *flow sinks*, i.e., flows ending within a component, and *flow paths*, i.e., flows through a component from its incoming ports to its outgoing ports.

(30) *Flow implementations* describe actual flow sequences representing flow specifications through components and sets of components across one or more connections. They are declared in component implementations. An *end-to-end flow* specifies a flow that starts within one subcomponent and ends within another subcomponent. Flow specifications, flow implementations, and end-to-end flows can have expected and actual values for flow related properties, e.g., latency or rounding error accumulation.

(31) An actual embedded system is represented by an instance of an AADL system implementation that consists of subcomponents representing the application software, the computing platform, and the physical environment.

(32) An AADL specification may be used in a variety of ways by a variety of tools during a broad range of life- cycle activities, e.g., for documentation during preliminary specification, for schedulability or reliability analysis during design studies and during verification, for generation of system integration code during implementation. Note that application software components must be bound to execution platform components - ultimately threads to processors and binary images to memory in order for the system to be analyzable for runtime properties and the actual system to be constructed from the AADL specification. Many uses of an AADL specification need not be fully automated, e.g., some implementation steps may be performed by hand.

(33) The AADL core language is extensible through property sets, *annex subclauses* and *annex libraries* that can be standardized or user-defined. Examples of standardized extensions are the Error Model Annex (see Annex Document C) and the Behavior Annex (Annex Document D). Property sets extend AADL by introducing additional properties that can be associated with elements of a model, while annexes introduce sublanguages that can be used to annotate a model. *Annex subclauses* consist of annex-specific sublanguages whose constructs can be added to component types and component implementations. *Annex libraries* are declarations of reusable annex-specific sublanguage elements that are placed in AADL packages and can be referenced in annex subclauses.

4 Components, Packages, and Annexes

(1) The AADL defines the following *categories* of components: *data, subprogram*, *subprogram group*, *thread, thread group, process, memory, bus, virtual bus*, *processor, virtual processor, device, system*, and *abstract*. The category *abstract* can be refined into any of the other categories. This section describes those aspects of components that are common to all AADL component categories. This section also describes packages as an organizing mechanism for component types and implementations. This section closes with the definition of annex subclauses and annex libraries.

(2) A component represents some hardware or software entity that is part of a system being modeled in AADL. A component has a *component type*, which defines a functional interface. The component type acts as the specification of a component that other components can operate against. It consists of features, flows, and property associations.

(3) A feature models a characteristic of a component that is visible to other components. Features are named, externally visible parts of the component type, and are used to exchange control and data via connections with other components. Features include ports to support directional flow of data and control, and subprograms including support for remote procedure call interactions. Features define parameters that represent the data values that can be passed into and out of subprograms. Features specify component access requirements for external data and bus components. Component access requirements for external subprograms support remote procedure call interactions.

(4) A component has zero or more *component implementations.* Component implementations represent variants of a component that adhere to the same interface, but may have different property values. A component implementation specifies the realization of a component variant, i.e., an internal structure for a component as an assembly of subcomponents. Subcomponents are instantiations of *component classifiers*, i.e., component types and implementations.

(5) Components are named and have properties. These properties have associated values that represent attributes and characteristics of a component.

(6) Components can be declared in terms of other components by refining and extending existing component types and component implementations. This permits partially complete component type and implementation declarations to act as templates that may have explicit parameter (*prototype*) specifications. Such templates can represent a common basis for the evolution of a family of related component types and implementations.

(7) This standard defines basic concepts and requirements for determining compliance between a component specification and a physical component. Within this framework, annexes to this standard will specify detailed compliance requirements for specific software programming, application modeling, and hardware description languages. This standard does not restrict the lower-level representation(s) used for software components, e.g., binary images, conventional programming languages, application modeling languages, nor does it restrict the lower-level representation(s) used for physical hardware component designs, e.g., circuit diagrams, hardware behavioral descriptions.

4.1 AADL Specifications

(1) An AADL specification consists of package and property set declarations that are considered to be global declarations.

(2) Packages provide a way for organizing collections of component type and implementation, feature group type, annex library declarations along with relevant property associations.

(3) Property sets provide extensions to the core AADL in the form of new properties and property types in support of additional modeling and analysis capabilities. Similarly annex libraries and annex subclauses provide extension to the core AADL to annotate the base model through the use of sublanguages.

(4) The content of packages, e.g., classifiers, can be referenced from anywhere by qualifying the classifier reference with the package name. The content of property sets, i.e., property type, property constant and property definitions, can be referenced by qualifying the property type, constant, or property reference with the property set name. References to predeclared properties do not require qualification by property set. Classifiers in other packages and properties in property sets can only be referenced with those packages and property sets are listed in the **with** declaration of a package or property set.

(5) System instances are identified to processing tools and methodologies by referencing a system implementation component as the root of the system instance (see Section 13.1).

Syntax

```
AADL_specification ::=
    package_spec | property_set
```

Naming Rules

(N1) An AADL specification has one *global* namespace. The package and property set identifiers reside in this space and must be unique.

(N2) These package and property set identifiers qualify the names of individual elements contained in them when they are referenced.

(N3) Package declarations represent labeled namespaces for component type, component implementation, feature group type, and annex library declarations.

(N4) Property set declarations represent labeled namespaces for property type and property definition declarations.

(N5) Packages and property sets may be separately stored. Those packages and property sets are considered to be part of the global namespace.

(N6) Defining identifiers in AADL must not be one of the reserved words of the language (see Section 15.7).

(N7) The AADL identifiers and reserved words can be in upper or lower case (or a mixture of the two) (see Section 15).

(N8) The AADL does not require that an identifier be declared before it is referenced.

Semantics

(6) An AADL specification provides a global namespace for packages and property sets. Items in packages and property sets can refer to items in other packages and property sets.

(7) A package is used to introduce new component types, component implementations, feature group types, and annex libraries.

(8) A property set is used to introduce new property types and properties (see section 11.1). They extend the predefined set of properties of the core AADL.

(9) Declarations in an AADL specification can refer to packages and property sets declared in separately stored AADL specifications. This allows packages and property sets to be stored separately and used by multiple AADL specifications. Mechanisms for locating such separately declared packages and property sets are tool specific.

Processing Requirements and Permissions

(10) A method of processing must accept an AADL specification presented as a single string of text in which declarations may appear in any order. An AADL specification may be stored as multiple pieces of specification text that are named or indexed in a variety of ways, e.g., a set of source files, a database, a project library. Preprocessors or other forms of automatic generation may be used to process AADL specifications to produce the required specification text. This approach makes AADL scalable in handling large models.

4.2 Packages

(1) A package provides a way to organize component types, component implementations, feature group types, and annex libraries into related sets of declarations by introducing separate namespaces. Package names are built using identifiers that are separated by double colons ("::"). This avoids the problem of duplicate names which might occur when packages are developed independently and then combined to model an integrated system. In other words, `complete_sys::first_independent::fuel_flow` is distinct from `complete_sys::second_independent::fuel_flow`. Packages cannot be declared inside other packages.

(2) Packages have public and private sections. Classifier declarations in public sections are accessible to other packages, while classifiers in private sections can only be referenced within the private section of the same package. This allows classifiers used in the implementation of public components to be kept private.

(3) Visibility to other packages is specified by a **with** declaration. Only classifiers in the packages listed in the **with** declaration can be referenced from within a given package. A **renames** declaration can be used to introduce a local alias to a qualified component type and feature group type reference.

(4) Visibility to property sets is specified by a **with** declaration. Only properties, property types, and property constants defined in the property sets listed in the **with** declaration can be referenced from within a given package unless they are predeclared properties, property types, or property constants.

Syntax

```
package_spec ::=
    package defining_package_name
     ( public package_declarations [ private package_declarations ]
       | private package_declarations )
    [ properties ( { property_association }+ |
      none_statement ) ]
end defining_package_name ;


package_declarations ::=
    { name_visibility_declaration }* { AADL_declaration }*


package_name ::=
    { package_identifier :: }* package_identifier


none_statement ::= none ;


AADL_declaration ::=
    classifier_declaration
    | annex_library


classifier_declaration ::=
    component_classifier_declaration | feature_group_classifier_declaration
```

```
component_classifier_declaration ::=
    component_type | component_type_extension |
    component_implementation | component_implementation_extension

feature_group_classifier_declaration ::=
        feature_group_type | feature_group_type_extension

name_visibility_declaration ::=
        import_declaration |
        alias_declaration

import_declaration ::=
       with ( package_name | property_set_identifier )
            { , ( package_name | property_set_identifier ) }* ;

alias_declaration ::=
    ( defining_identifier renames package package_name ; ) |
    ( [ defining_identifier ] renames
      ( component_category unique_component_type_reference |
      feature group unique_feature_group_type_reference ) ; ) |
    ( renames package_name::all ; )
```

NOTES:

The **properties** subclause of the package is optional, or requires an explicit empty subclause declaration. The latter is provided to accommodate AADL modeling guidelines that require explicit documentation of empty subclauses. An empty subclause declaration consists of the reserved word of the subclause and a none statement (**none ;**).

Naming Rules

(N1) A defining package name consists of a sequence of one or more package identifiers separated by a double colon ("::"). A defining package name must be unique in the global namespace.

(N2) The **public** and **private** section of a package may be declared in separate package declarations; these two declarations introduce a single defining package name.

(N3) Associated with every package is a *package namespace* that contains the names for all the elements defined within that package. This means that component types, feature group types, and defining entities declared in an annex library using an annex-specific sublanguage can be declared with the same name in different packages.

(N4) The package namespace is divided into a **public** section and a **private** section. Items declared in the **public** section of the package namespace can be referenced from outside the package as well as within the package.

(N5) The reference to an item declared in another package must be an item *name* qualified with a package name separated by a double colon ("::"). The package name must be listed in the `import_declaration` (**with**) of the package with the reference, i.e., in the `import_declaration` of the public section, if the reference is in the public section, and in the `import_declaration` of the public or private section if the reference is in the private section of the package. Only classifiers in the public package section can be referenced.

(N6) The reference to a property other than predeclared properties must be an property *name* qualified with a property set name separated by a double colon ("::"). The property set name must be listed in the **with** declaration of the package with the reference. Predeclared properties may be, but are not required to be qualified by the property set name.

(N7) The package name in a `import_declaration` must exist in the global name space.

(N8) The property set identifier in a `import_declaration` must exist in the global name space.

(N9) Items declared in the **private** section of the package can only be referenced from within the **private** section of the package.

(N10) If the qualifying package identifier of a qualified reference is missing, the referenced component classifier, feature group type, or item in an annex library must exist in the same package as the reference.

(N11) The package name referenced in an `alias_declaration` must exist in the global namespace and must be listed in the `import_declaration`.

(N12) The classifier referenced by the `alias_declaration` must exist in the name space of the **public** section of the package being referenced by the `alias_declaration`.

(N13) The classifier referenced by the alias declaration must refer to a component type or a feature group type.

(N14) The defining identifier of a `alias_declaration` must be unique in the namespace of the package containing the `alias_declaration`.

(N15) The `alias_declaration` makes the publicly visible identifier of classifiers declared in another package accessible in the name space of the package containing the `alias_declaration`. If the alias declaration for a classifier does not include a defining identifier then the referenced classifier identifier is used as defining identifier and this identifier must be unique in the namespace of the package with the alias declaration.

(N16) If the `alias_declaration` renames all publicly visible identifiers of component types and feature group types by naming the package and **all**, then all those identifiers must also be unique in the namespace of the package with the alias declaration.

(N17) The identifiers introduced by the `alias_declaration` are only accessible within the package. When declared in the public package section, they can be referenced within the public and private package section, but not from other packages. When declared in the private package section, they can be referenced within the private package section only.

Legality Rules

(L1) The defining package name following the reserved word **end** must be identical to the defining package name following the reserved word **package**.

(L2) For each package there may be at most one **public** section declaration and one **private** section declaration. These two sections may be declared in a single package declaration or in two separate package declarations.

(L3) A component implementation may be declared in both the public and private part of a package. In that case the declaration in the public part may only contain a **properties** subclause and a **modes** subclause.

Semantics

(5) A package provides a way to organize component type declarations, component implementation declarations, feature group types, and annex libraries into related sets of declarations along with relevant property associations. It provides a namespace for component types, feature group types, and annex libraries with the package name acting as a qualifier. Nested package names allow for unique package naming conventions that address potential name conflicts in component type and implementation names when independently developed AADL specifications are combined. Note that component implementations are named relative to component types. Thus, qualified component type names act as unique qualifier for component implementation names. Packages can be organized hierarchically by giving them nested package names. These package names represent absolute paths from the root of the package hierarchy.

(6) Packages have a **public** and a **private** section. Declarations in the **public** section are visible outside the package, i.e., names declared in the **public** part can be referenced by declarations in other packages. Declarations in the **private** section are visible only within **private** section of the package, i.e., they cannot be referenced from the **public** section or from other packages.

(7) Component type and component implementation declarations model execution platform and application software components of a system. A component type denotes externally visible characteristics of a component, i.e., its features and its properties. A component implementation denotes the internal structure, operational modes, and properties of a component. A component type can have several component implementations. This can be used for example to model product line architectures running on different execution platforms. Packages allow such declarations to be organized into separate namespaces.

(8) Feature group types provide the definition of an interface to a component that represents a collection of features or feature groups defined within the component implementation (see Section 5.3). This group of features may be accessed externally as a single unit.

(9) A component implementation can be declared in both the public and private section of a package. If it is declared in both, then the public declaration is limited to containing property associations and modes and only those items are visible outside the package. This allows component implementations to be made visible to other packages as variants of the same component type, while the details of the implementation, i.e., its realization expressed by the subcomponents and connections subclauses, are hidden in the private part. The two declarations represent the same component implementation.

(10) An `import_declaration` specifies which packages and property sets can be named in qualified references to items in other packages or property sets. Packages can be declared with `import_declaration` without classifiers to set up an initial collection of package with use restrictions on other packages.

(11) An `alias_declaration` introduces local identifiers as short names for long names. It does so for package names and for classifier type references qualified by a package name. The short name may differ from the identifier of the long name to avoid name conflicts.

(12) Property associations declared in the properties section of a package are associated with the package.

Examples

```
package Aircraft::Cockpit
public
   with Avionics::DataTypes, Safety_Properties;
   AirData renames Avionics::DataTypes::AirData;
   system MFD
   features
      Airdata: in data port AirData;
   properties
```

```
      Safety_Properties::Safety_Criticality => high;
   end MFD;
end Aircraft::Cockpit;
```

4.3 Component Types

(1) A component type specifies the external interface of a component that its implementations satisfy. It contains declarations that represent features of a component and property associations. Features of a component are ports, feature groups, required access to externally provided data, subprogram, and bus components, and parameter declarations for the specification of the data values that flow into and out of subprograms. The ports and feature groups of a component can be connected to compatible ports or subprograms of other components through connections to represent control and data interaction between those components. Required access to an external subcomponent, such as data, subprogram, or bus, is connected via required and provided access to a subcomponent of the specified component classifier.

(2) Component types can declare flow specifications, i.e., logical flows of information from its incoming ports to its outgoing ports that are realized by their implementations.

(3) Component types can declare modes and mode transitions of a component. These modes and mode transitions are common to all implementations of the component. This allows for mode-specific property values to be associated with the component type, its features, and its flows.

(4) Component types can be declared in terms of other component types, i.e., a component type can *extend* another component type – inheriting its declarations and property associations. If a component type extends another component type, then features, flows, and property associations can be added to those already inherited. A component type extending another component type can also refine the declaration of inherited feature and flow declarations by more completely specifying partially declared component classifiers of features and by associating new values with properties. A component type cannot be an extension of multiple component types, i.e., multiple inheritance is not supported for AADL components.

(5) Component types can declare *prototypes*, i.e., classifier parameters that are used in features. The prototype bindings are supplied when the component types is being extended or used in subcomponent declarations.

(6) Component type extensions form an *extension hierarchy*, i.e., a component type that extends another component type can also be extended. We use AADL graphical notation (see Appendix D) to illustrate the extension hierarchy in Figure 2. For example, component type GPS extends component type Position System inheriting ports declared in Position System. It may add a port, refine the data type classifier of a port incompletely declared in Position System, and overwrite the value of one or more properties. Component types being extended are referred to as *ancestors*, while component types extending a component type are referred to as *descendents*.

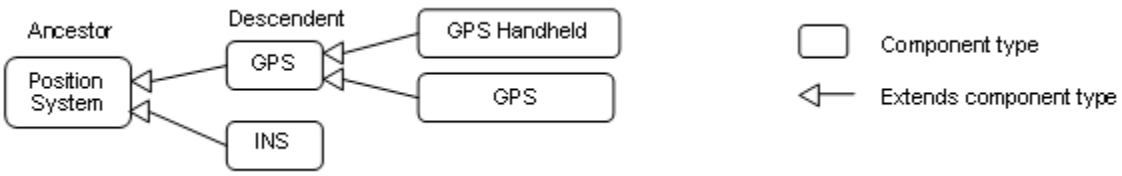


Figure 2 Component Type Extension Hierarchy

(7) Component types may also be extended using an `annex_subclause` to specify additional characteristics of the type that are not defined in the core of the AADL (see Section (18))

Syntax

```
component_type ::=
  component_category defining_component_type_identifier
  [ prototypes ( { prototype }+ | none_statement ) ]
  [ features ( { feature }+ | none_statement ) ]
  [ flows ( { flow_spec }+ | none_statement ) ]
  [ modes_subclause | requires_modes_subclause ]
  [ properties (
    { component_type_property_association | contained_property_association  }+
    | none_statement ) ]
  { annex_subclause }*
  end defining_component_type_identifier ;

component_type_extension ::=
  component_category defining_component_type_identifier
    extends unique_component_type_reference [ prototype_bindings ]
  [ prototypes ( { prototype | prototype_refinement }+ | none_statement ) ]
  [ features ( { feature | feature_refinement }+ | none_statement ) ]
  [ flows ( { flow_spec | flow_spec_refinement }+ | none_statement ) ]
  [ modes_subclause | requires_modes_subclause ]
  [ properties (
    { component_type_property_association | contained_property_association  }+
    | none_statement ) ]
  { annex_subclause }*
  end defining_component_type_identifier ;

component_category ::=
              abstract_component_category
            | software_category
            | execution_platform_category
            | composite_category

abstract_component_category ::= abstract

software_category ::= data | subprogram | subprogram group |
                      thread | thread group | process

execution_platform_category ::=
        memory | processor | bus | device | virtual processor | virtual bus
```

```
composite_category ::= system
```

```
unique_component_type_reference ::=
    [ package_name :: ] component_type_identifier
```

NOTES: The above grammar rules characterize the common syntax for all component categories. The sections defining each of the component categories will specify further restrictions on the syntax.

The **prototypes**, **features**, **flows**, **modes**, **annex**, and **properties** subclauses of the component type are optional, or require an explicit empty subclause declaration. The latter is provided to accommodate AADL modeling guidelines that require explicit documentation of empty subclauses. An empty subclause declaration consists of the reserved word of the subclause and a none statement (**none ;**).

Naming Rules

(N1) The defining identifier for a component type must be unique in the namespace of the package within which it is declared.

(N2) Each component type has a local namespace for defining identifiers of prototypes, features, modes, mode transitions, and flow specifications. That is, defining prototype, defining feature, defining modes and mode transitions, and defining flow specification identifiers must be unique in the component type namespace.

(N3) The component type identifier of the ancestor in a component type extension, i.e., that appears after the reserved word **extends**, must be defined in the specified package namespace. If no package name is specified, then the identifier must be defined in the namespace of the package the extension is declared in.

(N4) When a component type extends another component type, a component type namespace includes all the identifiers in the namespaces of its ancestors.

(N5) A component type that extends another component type does not include the identifiers of the implementations of its ancestors.

(N6) The defining identifier of a feature, flow specification, mode, mode transition, or prototype must be unique in the namespace of the component type.

(N7) The refinement identifier of a feature, flow specification, or prototype refinement refers to the closest refinement or the defining declaration of the feature going up the component type ancestor hierarchy.

(N8) The prototypes referenced by prototype binding declarations must exist in the local namespace of the component type being extended.

(N9) Mode transitions declared in the component type may not refer to event or event data ports of subcomponents.

Legality Rules

(L1) The defining identifier following the reserved word **end** must be identical to the defining identifier that appears after the component category reserved word.

(L2) The **prototypes**, **features**, **flows**, **modes**, and **properties** subclauses are optional. If a subclause is present but empty, then the reserved word **none** followed by a semi-colon must be present.

(L3) The category of the component type being extended must match the category of the extending component type, i.e., they must be identical or the category being extended must be **abstract**.

(L4) The classifier being extended in a component type extension may include prototype bindings. There must be at most one prototype binding for each prototype, i.e., once bound a prototype binding cannot be overwritten by a new binding in a component type extension.

(L5) A component type must not contain both a `requires_modes_subclause` and a `modes_subclause`.

(L6) If the extended component type and an ancestor component type in the extends hierachy contain modes subclauses, they must both be `requires_modes_subclause` or `modes_subclause`.

Standard Properties

```
Classifier_Substitution_Rule: inherit enumeration (Classifier_Match,  Type_Extension,
Signature_Match)
Prototype_Substitution_Rule: inherit enumeration (Classifier_Match, Type_Extension,
Signature_Match)
```

Semantics

(8) A component type represents the interface specification of a component, i.e., the component category, prototypes, features, flow specifications, modes, mode transitions, and property values of a component. A component implementation of this component type denotes a component, existing or potential, that is compliant with the component type declaration. Component implementations are expected to satisfy these externally visible characteristics of a component. The component type provides a contract for the component interface that users of the component can depend on.

(9) The component categories are: data, subprogram, subprogram group, thread, thread group, and process (software categories); processor, virtual processor, bus, virtual bus, memory, and device (execution platform categories); system (compositional category), and abstract component (compositional category). The semantics of each category will be described in sections 5, 6, and 7.

(10) Features of a component are interaction points with other components, i.e., ports and feature groups; subprogram parameters; data component access, subprogram access, and bus access. Ports represent directional flow of data and events between components, feature groups represent groups of features that are connected to another component, data component access represents access to shared data components, subprogram access represents access to a subprogram by a caller, and bus access represents access to a bus from processor, memory, device, and other bus components to establish hardware connectivity. Features are further described in Section 8.

(11) Flow specifications indicate whether a flow of data or control originates within a component, terminates within a component, or flows through a component from one of its incoming ports to one of its outgoing ports.

(12) Mode declarations define modes of the component that are common to all implementations. As a result, component types can have mode-specific property values. Other components can initiate mode transitions by supplying events to incoming event ports of a component. Mode transitions specify which event ports affect their transition. A component type extension can add modes and associate properties to existing modes.

(13) A `requires_modes_subclause` specifies as set of modes that the component expects to inherit from its containing component. In this case, the component can utilize these modes for mode-specific property values and for **in modes** declarations of subcomponents and connection, but mode transition behavior is determined by the containing component, whose modes are made accessible to the component.

(14) A component type can contain *incomplete* feature declarations, i.e., declarations with no *component classifier references* or just the component type name for a component type with more than one component implementation. The component implementation may not exist yet or one of several implementations may have not been selected yet.

(15) A component type may also be declared with *prototypes*, indicating that the component type is incomplete and can be parameterized. Classifiers and features can be supplied through prototype bindings to complete such a component type template as part of a component type extension, component implementation extension, or when used as a component type reference, e.g., in feature declarations or subcomponent declarations. Once bound a prototype cannot be rebound. The use of incomplete declarations is particularly useful during early design stages where details may not be known or decided.

(16) A component type can be declared as an extension of another component type resulting in a component type extension hierarchy, as illustrated in Figure 2. A component type extension can refine an abstract component type to one of the concrete component categories. A component type extension inherits the features, flow specifications, modes, mode transitions, prototypes, and properties of the component type being extended. For annex subclauses each annex defines whether annex declarations are inherited. A component type extension can contain refinement declarations to permit incomplete feature declarations to be completed and new property values to be associated with features and flow specification declared in a component type being extended. In addition, a type extension can add feature declarations, flow specifications, modes, mode transitions, and property associations.

(17) A component type being extended may include prototype binding declarations. If prototype bindings are declared for a subset of the prototypes, then only the prototypes without binding can be bound when referencing the component type extension. This supports evolutionary development and modeling of system families by declaring partially complete component types that get refined in extensions.

(18) Properties are predefined for each of the component categories and will be described in the appropriate sections. See Section 11.3 regarding rules for determining property values.

Examples

```
system File_System
features
   -- access to a data component
   root: requires data access FileSystem::Directory.hashed;
end File_System;

process Application
features
   -- a data out port
   result: out data port App::result_type;
   home: requires data access FileSystem::Directory.hashed;
end Application;

thread Calculate
prototypes
  -- A data type to be used as type for the input and result port
  data_type: data;
features
   input: in data port prototype data_type;
   result: out data port prototype data_type;
end Calculate;
```

```
thread Compute_Distance extends Calculate (data_type => App::Distance)
end Compute_Distance;
```

4.4 Component Implementations

(1) A *component implementation* represents the realization of a component in terms of subcomponents, their connections, flow sequences, properties, component modes and mode transitions. Flow sequences represent implementations of flow specifications in the component type, or end-to-end flows to be analyzed. Modes represent alternative operational modes that may manifest themselves as alternate configurations of subcomponents, connections, call sequences, flow sequences, and property values.

(2) A component type can have zero, one, or multiple component implementations. If a component type has zero component implementations, then it is considered to be a leaf in the system component hierarchy. For example, a partial AADL model may have processes as components without realization, while a task level AADL model expand to threads as leaves. If no implementation is associated then the properties on the component types provide information about the component for analysis and system generation.

(3) A component implementation can be declared as an extension of another component implementation. In that case, the component implementation inherits the declarations of its ancestors as well as its component type. A component implementation extension can refine inherited declarations, and add subcomponents, connections, subprogram call sequences, flow sequences, mode declarations, and property associations.

(4) Component implementations can declare *prototypes*, i.e., classifier parameters that are used in subcomponent declarations. The prototype bindings are supplied when the component implementation is being extended or used in subcomponent declarations.

(5) Component implementations build on the component type *extension hierarchy* in two ways. First, a component implementation is a realization of a component type (shown as dashed arrows in Figure 3). As such it inherits features and property associations of its component type and any component type ancestor. Second, a component implementation declared as extension inherits subcomponents, connections, subprogram call sequences, flow sequences, modes, property associations, and annex subclauses from the component implementation being extended (shown as solid arrows in Figure 3). A component implementation can extend a component implementation that in turn extends another component implementation, e.g., in Figure 3 *GPS*. Handheld extends *GPS.Basic* and is extended by *GPS_Secure.Handheld*. Component implementations higher in the extension hierarchy are called ancestors and those lower in the hierarchy are called descendents. A component implementation can extend another component implementation of its own component type, e.g., *GPS.Handheld* extends *GPS.Basic*, or it can extend the component implementation of one of its ancestor component types, e.g., *GPS_Secure.Handheld* extends *GPS.Handheld*, which is an implementation of the ancestor component type *GPS*. The component type and implementation extension hierarchy is illustrated in Figure 3.

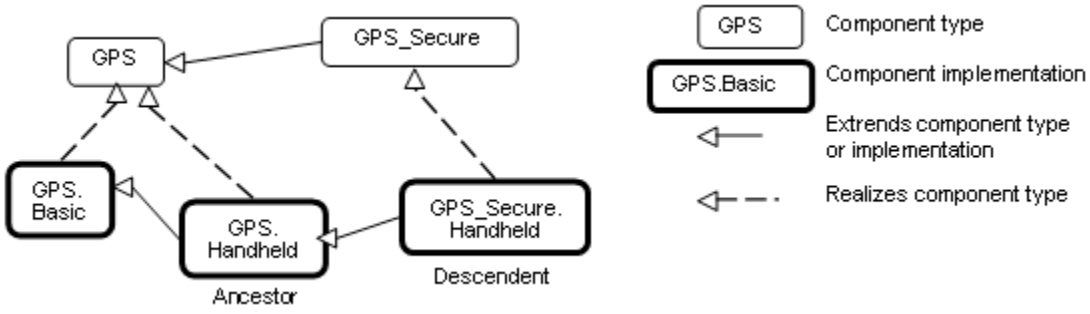


Figure 3 Extension Hierarchy of Component Types and Implementations

(6) A component implementation may also be extended using an `annex_subclause` to specify additional characteristics of the type that are not defined in the core of the AADL (see Section (18)).

Syntax

```
component_implementation ::=
   component_category implementation
   defining_component_implementation_name
  [ prototypes ( { prototype }+ | none_statement ) ]
   [ subcomponents ( { subcomponent }+  | none_statement ) ]
   [ calls ( { subprogram_call_sequence }+  | none_statement ) ]
   [ connections ( { connection }+ | none_statement ) ]
   [ flows ( { flow_implementation |
               end_to_end_flow_spec }+ | none_statement ) ]
   [modes_subclause ]
   [ properties ( { property_association | contained_property_association  }+
                  | none_statement ) ]
   { annex_subclause }*
   end defining_component_implementation_name ;

component_implementation_name ::=
        component_type_identifier . component_implementation_identifier

component_implementation_extension ::=
   component_category implementation
   defining_component_implementation_name
   extends unique_component_implementation_reference [ prototype_bindings ]
   [ prototypes ( { prototype | prototype_refinement }+ | none_statement ) ]
   [ subcomponents
        ( { subcomponent | subcomponent_refinement }+ | none_statement ) ]
   [ calls ( { subprogram_call_sequence }+  | none_statement ) ]
   [ connections
        ( { connection | connection_refinement }+  | none_statement ) ]
   [ flows ( { flow_implementation  | flow_implementation_refinement |
               end_to_end_flow_spec | end_to_end_flow_spec_refinement }+
               | none_statement ) ]
   [modes_subclause ]
   [ properties ( { property_association | contained_property_association }+
     | none_statement ) ]
   { annex_subclause }*
   end   defining_component_implementation_name ;

unique_component_implementation_reference ::=
    [ package_name :: ] component_implementation_name
```

NOTES: The above grammar rules characterize the common syntax for all component categories. The sections defining each of the component categories will specify further restrictions on the syntax.

The **prototypes**, **subcomponents**, **connections**, **calls**, **flows**, **modes**, and **properties** subclauses of the component implementation are optional or if used and empty, require an explicit empty declaration. The latter is provided to accommodate AADL modeling guidelines that require explicit documentation of empty subclauses. An empty subclause declaration consists of the reserved word of the subclause and a none statement (**none ;**).

The `annex_subclause` of the component implementation is optional.

The **refines type** subclause of AADL V1 has been removed. Its role was to allow association of implementation-specific property values to features and flow specifications declared in the component type. The same can be achieved by contained property associations whose **applies to** subclause names the feature or flow specification.

Naming Rules

(N1) A component implementation name consists of a component type identifier and a component implementation identifier separated by a dot (“.”). The first identifier of the defining component implementation name must name a component type that is declared in the same package as the component implementation, or name an *alias* to a component type in another package.

(N2) The defining identifier of the component implementation must be unique within the local namespace of the component type.

(N3) Every component implementation defines a *local namespace* for all defining identifiers of prototypes, subcomponents, subprogram calls, connections, flows, and modes declared within the component implementation. The defining identifier of a prototype, subcomponent, subprogram call, connection, flow, or mode must be unique within this namespace. For example, a subcomponent and a mode cannot have the same defining identifier within the same component implementation, or a mode with the same defining identifier cannot be declared in a component type and a component implementation.

(N4) This local namespace inherits the namespace of the associated component type, i.e., defining identifiers must be unique within the local namespace and also within the component type namespace.

(N5) Refinement identifiers of features must exist in the namespace of the associated component type or one of the component type’s ancestors. Refinement identifiers of subcomponent and connection refinements must exist in the local namespace of an ancestor component implementation.

(N6) In a component implementation extension, the component type identifier of the component implementation being extended, which appears after the reserved word **extends**, must be the same as or an ancestor of the component type of the extension. The component implementation being extended may exist in another package. In this case the component implementation name is qualified with the package name.

(N7) When a component implementation **extends** another component implementation, the local namespace of the extension is a superset of the local namespace of the ancestor. That is, the local namespace of a component implementation inherits all the identifiers in the local namespaces of its ancestors (including the identifiers of their respective component type namespaces).

(N8) Within the scope of the component implementation, subcomponent declarations, connections, subprogram call sequences, mode transitions, and property associations can refer directly to identifiers in the local namespace, i.e., to declared prototypes, subcomponents, connections, and modes, as well as to required bus, data, subprogram, and subprogram group subcomponents and features declared in the associated component type.

(N9) The prototype referenced by the prototype binding declaration must exist in the local namespace of the component implementation being extended. In other words, prototype binding declarations may bind prototypes of the component type and of the component implementation.

NOTES:

A component type can be made visible in a new namespace (package) without qualifying it with the package name by declaring it as **renames** that refers to the component type in the original package. This allows an implementation to be placed in a separate package from the component type package. In the following example, **renames** LM::GPS allows the type GPS to be named without a qualifying package name. This allows a system implementation to be added to GPS in the package LM1.

package LM1 **public**

renames LM::GPS;

system implementation GPS.newimpl

Legality Rules

(L1) The pair of identifiers separated by a dot (".") following the reserved word **end** must be identical to the pair of identifiers following the reserved word **implementation**.

(L2) The **prototypes**, **subcomponents**, **connections**, **calls**, **flows**, **modes**, and **properties** subclauses are optional. If they are present and the set of feature or required subcomponent declarations or property associations is empty, **none** followed by a semi-colon must be present in that subclause.

(L3) The category of the component implementation must be identical to the category of the component type for which the component implementation is declared.

(L4) If the component implementation extends another component implementation, the category of both must match, i.e., they must be identical or the category being extended must be **abstract**.

(L5) The classifier being extended in a component implementation extension may include prototype bindings. There must be at most one prototype binding for each unbound prototype.

(L6) If the component type of the component implementation contains a `requires_modes_subclause` then the component implementation must not contain any modes subclause.

(L7) If modes are declared in the component type, then modes cannot be declared in component implementations.

(L8) If modes or mode transitions are declared in the component type, then mode transitions can be added in the component implementation. These mode transitions may refer to event or event data ports of the component type and of subcomponents.

(L9) The category of a subcomponent being refined must match the category of the refining subcomponent declaration, i.e., they must be identical or the category being refined must be **abstract**.

(L10) For all other refinement declarations the categories must match (see the respective sections).

Standard Properties

`Classifier_Substitution_Rule: inherit enumeration (Classifier_Match, Type_Extension, Signature_Match)`

`Prototype_Substitution_Rule: inherit enumeration (Classifier_Match, Type_Extension, Signature_Match)`

`Classifier_Matching_Rule: inherit enumeration (Classifier_Match, Equivalence, Subset, Conversion, Complement)`

Semantics

(7) A component implementation represents the internal structure of a component through subcomponent declarations. Interaction between subcomponents is expressed by the connections, flows, and subprogram call sequences. Mode declarations represent alternative runtime configurations (internal structure) and alternative execution behavior (interaction between subcomponents). A component implementation also has property values to express its non-functional attributes such as safety level or execution time which can also vary by mode.

(8) Each component implementation is associated with a component type and provides a realization of its features (interface). A component type can have multiple implementations. A component implementation can be viewed as a component variant with the same interface but with differing property values that characterize the differences between implementations without exposing the internals of their realization. This can be achieved by placing the component implementation declaration without subcomponents and connections in the public section of an AADL package, and a component implementation declaration with subcomponents and connections for the same component in the private section of an AADL package.

(9) The actual system being modeled by component types and component implementations may contain subcomponents, some of which may contain subcomponents themselves. The subcomponent containment hierarchy reflects the actual system structure.

(10) A component implementation that is an extension of another inherits all prototypes, subcomponents, connections, subprogram call sequences, flow sequences (flow implementations and end-to-end flows), modes, mode transitions, and property associations from its ancestors as well as features, and associations, from its associated component type (and that component type's ancestors). For annex subclauses each annex defines whether annex declarations are inherited.

(11) A component implementation extension can refine prototypes and subcomponents previously declared in ancestor component implementations by supplying component classifiers, and by associating new property values. A component implementation extension can refine connections, flows, and modes of its ancestor component implementations by associating new property values. A component implementation extension can refine features of its associated component type (and that component type's ancestors) by associating new property values to them. A component implementation being extended may include declaration of prototype bindings. If prototype bindings are declared for a subset of the prototypes, then only the prototypes without binding can be bound in component implementation extensions.

(12) A component implementation extension can also add prototypes, subcomponents, connections, subprogram call sequences, flow sequences, modes, property associations, and annex subclauses. This extension capability supports evolutionary development and modeling of system families by declaring partially complete component implementations that get refined in extensions.

(13) A descendent component implementation is said to contain all subcomponents whose identifiers appear in its local namespace, i.e., subcomponents declared in the component implementation and any of its ancestors. In other words, an instance of a component implementation extension contains instances of declared and inherited subcomponents, features, connections, subprogram call sequences, flow sequences, and modes.

(14) Properties are predefined for each of the component categories and will be described in the appropriate sections. See Section 11.3 regarding rules for determining property values.

Processing Requirements and Permissions

(15) A component implementation denotes a set of actual system components, existing or potential, that are compliant with the component implementation declaration as well as the associated component type. That is, the actual components denoted by a component implementation declaration are always compliant with the functional interface specified by the associated component type declaration. Actual components denoted by different implementations for the same component type differ in additional details such as internal structure or behaviors; these differences may be specified using properties.

(16) In general, two actual components that comply with the same component type and component implementation are not necessarily substitutable for each other in an actual system. This is because an AADL specification may be legal but not specify all of the characteristics that are required to ensure total correctness of a final assembled system. For example, two different versions of a piece of source text might both comply with the same AADL specification, yet one of them may contain a programming defect that results in unacceptable runtime behavior. Compliance with this standard alone is not sufficient to guarantee overall correctness of a actual system.

Examples

```
data Bool_Type
end Bool_Type;

thread DriverModeLogic
features
   BreakPedalPressed : in data port Bool_Type;
   ClutchPedalPressed : in data port Bool_Type;
   Activate : in data port Bool_Type;
   Cancel : in data port Bool_Type;
   OnNotOff : in data port Bool_Type;
   CruiseActive : out data port Bool_Type;
end DriverModeLogic;

-- Two implementations whose source texts use different variable names for
-- their cruise active port
thread implementation DriverModeLogic.Simulink
properties
  Dispatch_Protocol=>Periodic;
  Period=> 10 ms;
  Source_Name => "CruiseControlActive" applies to CruiseActive;
end DriverModeLogic.Simulink;

thread implementation DriverModeLogic.C
properties
  Dispatch_Protocol=>Periodic;
  Period=> 10 ms;
  Source_Name => "CCActive" applies to CruiseActive;
end DriverModeLogic.C;
```

4.5 Subcomponents

(1) A *subcomponent* represents a component contained within another component, i.e., declared within a component implementation. Subcomponents contained in a component implementation may be instantiations of component implementations that contain subcomponents themselves. This results in a component containment hierarchy that ultimately describes the whole actual system as a system instance. Figure 4 provides an illustration of a containment hierarchy using the graphical AADL notation (see Appendix D). In this example, Sys1 represents a system. The implementation of the system contains subcomponents named C3 and C4. Component C3, a subcomponent in Sys1's implementation, contains subcomponents named C1 and C2. Component C4, another subcomponent in Sys1's implementation, contains a second set of subcomponents named C1 and C2. The two subcomponents named C1 and those named C2 do not violate the unique name requirement. They are unique with respect to the local namespace of their containing component's local namespace.

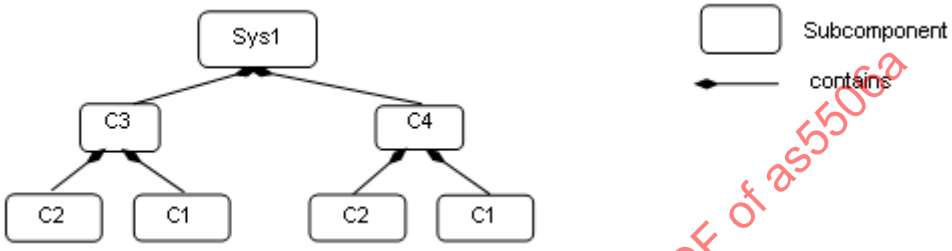


Figure 4 Component Containment Hierarchy

(2) A subcomponent declaration may resolve required subcomponent access declared in the component type of the subcomponent. For details on required subcomponent access see Section 8.4.

(3) A subcomponent can be declared to apply to specific modes (rather than all modes) defined within the component implementation.

(4) Subcomponents can be refined as part of component implementation extensions. Refinement allows classifier references to be completed, abstract subcomponents to be refined into one of the concrete categories, and subcomponent property values to be associated. The resulting refined subcomponents can be refined themselves.

(5) An array of subcomponents can be declared to represent a set of subcomponents with the same component type. This array may have one or more dimensions.

Syntax

```
subcomponent ::=
    defining_subcomponent_identifier :
       component_category
     [ unique_component_classifier_reference [prototype_bindings]
       | prototype_identifier ]
      [ array_dimensions [ array_element_implementation_list ] ]
       [ { { subcomponent_property_association
            | contained_property_association }+ } ]
        [ component_in_modes ] ;


subcomponent_refinement ::=
    defining_subcomponent_identifier : refined to
       component_category
         [ unique_component_classifier_reference [ prototype_bindings ]
```

```
          | prototype_identifier ]
          [ array_dimensions [ array_element_implementation_list ] ]
        [ { { subcomponent_property_association
              | contained_property_association }+ } ]
         [ component_in_modes ] ;

unique_component_classifier_reference ::=
         ( unique_component_type_reference
         | unique_component_implementation_reference )

array_dimensions ::=
    { array_dimension  }+

array_dimension ::=
    [ [ array_dimension_size ] ]

array_dimension_size ::=
   numeral | unique_property_constant_identifier | unique_property_identifier


array_element_implementation_list ::=
    ( unique_component_implementation_reference
      { , unique_component_implementation_reference }* )

-- array selection used in contained proper association and references
array_selection_identifier ::=
    identifier array_selection

array_selection ::=
    { [ selection_range ] }+

selection_range ::=
    numeral [ .. numeral ]
```

NOTES:

The above grammar rules characterize the common syntax for subcomponent of all component categories. The sections defining each of the component categories will specify further restrictions on the syntax.

Naming Rules

(N1) The defining identifier of a subcomponent declaration placed in a component implementation must be unique within the local namespace of the component implementation that contains the subcomponent.

(N2) The defining identifier of a subcomponent refinement must exist as a defining subcomponent identifier in the local namespace of an ancestor component implementation.

(N3) The component type identifier or the component implementation name of a component classifier reference must exist in the package namespace.

(N4) The prototype identifier of a prototype reference must exist in the local name space of the component implementation.

(N5) The prototype referenced by the prototype binding declarations must exist in the local namespace of the component classifier being referenced.

(N6) The modes named in the **in modes** statement of a subcomponent must refer to modes in the component implementation that contains the subcomponent or its component type. The modes named in the **in modes** statement of a property association of a subcomponent must refer to modes of the subcomponent, or in the case of a contained property association to modes of the last component in the component path (see Section 11.3).

Legality Rules

(L1) The category of the subcomponent declaration must match the category of its corresponding component classifier reference or its prototype reference, i.e., they must be identical, or in the case of a classifier reference the referenced classifier category may be *abstract*.

(L2) The component classifier reference of a subcomponent declaration may include prototype bindings for a subset or all of the component classifier prototypes. This represents an unnamed component classifier extension of the referenced classifier.

(L3) In a subcomponent refinement declaration the component category may be refined from **abstract** to one of the concrete component categories. Otherwise the category must be the same as that of the subcomponent being refined.

(L4) The `Classifier_Substitution_Rule` property specifies the rule to be applied when a refinement supplies a classifier and the original subcomponent declaration already has a component classifier. This property can be applied to individual subcomponents or features, or it can be inherited from classifiers. The following rules are supported:

- `Classifier_Match`: The component type of the refinement must be identical to the component type of the classifier being refined. If the original declaration specifies a component implementation, then any implementation of that type can replace this original implementation. This is the default rule.
- `Type_Extension`: Any component classifier whose component type is an extension of the component type of the classifier in the subcomponent being refined is an acceptable substitute.
- `Signature_Match`: The component type of the refinement must match the signature of the component type of the classifier being refined.

(L5) In the case of a signature match, the component type of the subcomponent being refined must have a subset of the features of the component type in the refinement. The features are compared by name matching; the feature categories and direction (in data port, provides data access, etc.) must be the same and any feature classifier must match according to rules defined for `Classifier_Match`. In addition, if flow specifications are present in the component type being refined, then the component type of the refinement must have at least the same set of flow specifications. Flow specifications with the same name must have the same source and destination ports.

(L6) The component category and optional component classifier or prototype reference can be followed by a set of array dimensions to define the subcomponent as an array of actual subcomponents.

(L7) The array size specification for the dimensions is optional. In this case the array declaration is considered incomplete. If the size of the array dimension is specified it must be specified for all dimensions in the same declaration.

(L8) When refining a subcomponent array the number of dimensions of the array cannot be changed, but the array size can be specified for each dimension if it was not specified in the subcomponent declaration being refined.

(L9) When the subcomponent is declared as an array with array dimension sizes then a list of component implementations can be supplied, one for each element of the array. Different implementations of the same component type can be chosen. The number of elements in the list must correspond to the number of elements in the component array. In the case of multi-dimensional arrays, the list elements are assigned by incrementing the index of the last dimension first.

(L10) Selecting index ranges in one or more dimensions of an array is only possible if the size of the array for these dimensions is already defined. The index range of a dimension is from 1 to the size of the dimension. Specification of array index ranges is limited to the **applies to** subclause of contained property associations. Specification of a single array element is limited to the **applies to** subclause of contained property associations and to the values of **reference** properties.

Consistency Rules

(C1) The classifier of a subcomponent cannot recursively contain subcomponents with the same component classifier. In other words, there cannot be a cyclic containment dependency between components.

Standard Properties

```
Classifier_Substitution_Rule: inherit enumeration (Classifier_Match, Type_Extension, Signature_Match)
Acceptable_Array_Size: list of Size_Range
```

Semantics

(6) Subcomponents declared in a component implementation are considered to be contained in the component implementation. Contained subcomponents are instantiated when the containing component implementation is instantiated. Thus, the component containment hierarchy describes the hierarchical structure of the actual system.

(7) A component implementation can contain *incomplete* subcomponent declarations, i.e., subcomponent declarations with no component classifier references, or if the component classifier reference only consists of a component type name for a component type with more than one component implementation. A subcomponent declaration is also incomplete when it consists of the declaration of an array of subcomponents for which the array sizes are not specified. This is particularly useful during early design stages where details may not be known or decided. Such incomplete subcomponent declarations can be refined in component implementation extensions.

(8) A subcomponent declaration can be parameterized by referring to a prototype. In this case the component category and component classifier bound to the prototype is used when the system is instantiated.

(9) A subcomponent declaration can reference a component classifier with prototype bindings. The prototype binding can refer to other classifiers or to a prototype of the component type or implementation that contains the subcomponent. In the latter case, the prototype actual is passed down levels of the component hierarchy and effectively allows the system subcomponents to be configured from a higher level component.

(10) A component classifier reference with prototype bindings that refer to component classifiers effectively is an unnamed extension of the classifier being referenced. In other words, it could have been declared as a component type or component implementation extension with a new defining identifier and this identifier could have been referenced in the subcomponent declaration. Two unnamed component classifier extensions are not considered to be extensions of each other.

(11) The optional `in_modes` subclause specifies the modes in which the subcomponent is active. An `in_modes` in a subcomponent refinement replaces previously specified subsets of modes or all modes. A subcomponent refinement without `in_modes` specifies that the subcomponent is active in all modes. The `in_modes` refer to modes of the component implementation that contains the subcomponent or its component type.

(12) A subcomponent can have property values associated to itself, or a contained property association can be declared for one of the subcomponents in its containment hierarchy, as well as those subcomponents' features, modes, subprogram call sequences, connections, and flows, or model elements in any annex subclause of a subcomponent (see Section 11.3). Subcomponent refinements may declare property associations – that override the property values declared in the subcomponent being refined. Property associations can have `in_modes` statements that refer to modes of the component implementation that contains the subcomponent, or in the case of contained property associations also to modes of the last subcomponent named in the path of the **applies to** (see Section 11.3).

(13) The arrays of subcomponents are used to simplify the declaration of a multiplicity of subcomponents with the same classifier without declaring each of them separately. If a size of a subcomponent array is not known the array is incomplete and is assumed to have one element for the purpose of system instances of incomplete models. A subcomponent array can only be refined by adding array sizes to the dimensions if they are without a size.

(14) All elements of a subcomponent array have the same component classifier, i.e., they are of the same kind. A subcomponent array can also be declared to have the same component type, but its elements vary in their implementation, e.g., to represent variants in an N-Version redundancy pattern.

(15) A property association declared with a subcomponent array applies to each element in the array. Contained property associations declared in the enclosing component implementation can be used to associate different property values to different elements or subsets of the subcomponent array.

Processing Requirements and Permissions

(16) If the subcomponent declaration references a component type and the type has a single implementation then a method of processing (tool) is permitted to generate a complete system instance by choosing the single implementation even if it is not named. If the referenced component type has multiple implementations then the implementation must be explicitly referenced. However, some project may impose design constraints that require modelers to completely specify such classifier references.

Examples

(17) The example illustrates modeling of source text data types as data component types without any implementation details. It illustrates the use of **package** to group data component type declarations. It illustrates both component classifier references to component types and to component implementations. It illustrates the use of ports as well as required and provided data access, and required subprogram access. In that context it illustrates the ways of resolving required access. The Data Modeling Annex (Annex Document B) provides guidance on how to effectively represent data models of applications in AADL.

```
package Sampling
public
   data Sample
   properties
      Source_Data_Size => 16 Bytes;
   end Sample;

   data Sample_Set
   properties
      Source_Data_Size => 1 MByte;
   end Sample_Set;

   data implementation Sample_Set.impl
   subcomponents
```

```
      Data_Set: data Sample ;
   end Sample_Set.impl;

   data Dynamic_Sample_Set extends Sample_Set
   end Dynamic_Sample_Set;

   data implementation Dynamic_Sample_Set.impl extends Sample_Set.impl
   properties
      Source_Data_Size => 8 Bytes applies to Data_Set;
   end Dynamic_Sample_Set.impl;
end Sampling;

thread Init_Samples
features
   OrigSet : requires data access Sampling::Sample_Set;
   SampleSet : requires data access Sampling::Sample_Set;
end Init_Samples;

thread Collect_Samples
features
   Input_Sample : in event data port Sampling::Sample;
   SampleSet : requires data access Sampling::Sample_Set;
   Filtering_Routine: requires subprogram access Sample_Subprogram;
end Collect_Samples;

thread implementation Collect_Samples.Batch_Update
properties
   Source_Name => "InSample" applies to Input_Sample;
end Collect_Samples.Batch_Update;

thread Distribute_Samples
features
   SampleSet : requires data access Sampling::Sample_Set;
   UpdatedSamples : out event data port Sampling::Sample;
end Distribute_Samples;

process Sample_Manager
features
   Input_Sample: in event data port Sampling::Sample;
```

```
   External_Samples: requires data access Sampling::Sample_Set;
   Result_Sample: out event data port Sampling::Sample;
end Sample_Manager;

process implementation Sample_Manager.Slow_Update
subcomponents
   Samples: data Sampling::Sample_Set;
   Init_Samples : thread Init_Samples;
   -- the required access is resolved to a subcomponent declaration
   Collect_Samples: thread Collect_Samples.Batch_Update;
   Distribute: thread Distribute_Samples;
   Sample_Filter: subprogram Sample_Subprogram.Simple;
connections
   data access Samples <-> Init_Samples.SampleSet;
   data access External_Samples <-> Init_Samples.OrigSet;
   data access Samples <-> Collect_Samples.SampleSet;
   port Input_Sample -> Collect_Samples.Input_Sample;
   data access Samples <-> Distribute.SampleSet;
   port Distribute.UpdatedSamples -> Result_Sample;
   subprogram access Sample_Filter <-> Collect_Sample.Filtering_Routine;
end Sample_Manager.Slow_Update;
```

(18) This example illustrates the use of arrays in defining a triple redundancy pattern with a voter. The pattern is defined as an **abstract** component (see Section 4.6) that uses data ports. The connections are defined with a connection pattern property to indicate how the elements of the source array are connected to the destination. Each instance of `MyProcess` is connected to a separate port of the `Voter`. Note that the number of replicates could be kept flexible by specifying the array dimension size through a property.

```
package Redundancy
public
   abstract Triple
   features
     input: in data port;
     output: out data port;
   end Triple;

   abstract implementation Triple.impl
   subcomponents
     MyProcess: abstract Calculate [3];
     MyVoter: abstract Voter;
   connections
```

```
      extinput: port input -> MyProcess.input
                    { Connection_Pattern => One_To_Many; };
      tovoter: port MyProcess.output -> MyVoter.input
                    { Connection_Pattern => One_To_One; };
      extoutput: port MyVoter.output -> output;
   end Triple.impl;

   abstract Calculate
   features
     input: in data port;
     output: out data port;
   end Calculate;

   abstract Voter
   features
     input: in data port [3];
     output: out data port;
   end Voter;
end Redundancy;
```

4.6 Abstract Components

(1) The component category **abstract** represents an abstract component. Abstract components can be used to represent component models. Abstract component can contain any component and can be contained in any component. The abstract component category can later be refined into one of the concrete component categories: any of the software components, hardware components, and composite components. When an abstract component is refined into a concrete component category it must adhere to the containment rules imposed by the concrete category. For example, an abstract subcomponent of a process can only be refined into a thread or thread group.

Legality Rules

Category	Type	Implementation
abstract	Features: • port • feature group • provides data access • provides subprogram access • provides subprogram group access • provides bus access • requires data access • requires subprogram access • requires subprogram group access • requires bus access • (abstract) feature Flow specifications: yes Modes: yes Properties: yes	Subcomponents: • data • subprogram • subprogram group • thread • thread group • process • processor • virtual processor • memory • bus • virtual bus • device • system • abstract Subprogram calls: yes Connections: yes Flows: yes Modes: yes Properties: yes

(L1) An **abstract** component type declaration can contain feature declarations (including abstract feature declarations), flow declarations, as well as property associations.

(L2) An **abstract** component implementation can contain subcomponent declarations of any category. Certain combinations of subcomponent categories are only acceptable if they are acceptable in one of the concrete component categories.

(L3) An **abstract** component implementation can contain a modes subclause, a connections subclause, a flows subclause, and property associations.

(L4) An **abstract** subcomponent can be contained in the implementation of any component category.

(L5) If an **abstract** subcomponent is refined to a concrete category, the concrete category must be acceptable to the component implementation category whose subcomponent is being refined.

(L6) An **abstract** subcomponent can be declared as an array of subcomponents.

(L7) If an **abstract** component type is refined to a concrete category, the features of the abstract component type must be acceptable for the concrete component type.

(L8) If an **abstract** component implementation is refined to a concrete category, the subcomponents of the abstract component implementation must be acceptable for the concrete component implementation.

Standard Properties

(2) An **abstract** component can have property associations of properties that apply to any concrete category. However, when refined to a concrete category, properties that do not apply to the concrete category will be ignored. A method of processing may provide a warning about ignored properties.

Semantics

(3) The component of category **abstract** represents an abstract component. It can be used to represent conceptual architectures. This abstract component can be refined into a runtime architecture by refining the component category into a software, composite, or hardware component. Such a refinement from a conceptual architecture to a runtime architecture is illustrated in the example below.

(4) Alternatively, the conceptual architecture can be defined in terms of abstract components and the runtime architecture can be defined separately in terms of threads and processes. A user-defined property of the **reference** property type can be used to specify the mapping of conceptual components to runtime architecture components.

Examples

(5) A conceptual architecture and its refinement into a runtime architecture.

```
abstract car
end car;

abstract implementation car.generic
subcomponents
  PowerTrain: abstract power_train;
  ExhaustSystem: abstract exhaust_system;
end car.generic;

abstract power_train
features
  exhaustoutput: requires bus access Manifold;
end power_train;

abstract exhaust_system
features
  exhaustManifold: provides bus access Manifold;
end exhaust_system;

-- runtime architecture
system carRT extends car
end carRT;

system implementation carRT.impl
   extends car.generic
subcomponents
    PowerTrain : refined to process power_train;
    ExhaustSystem : refined to process exhaust_system;
end carRT.impl;
```

4.7 Prototypes

(1) Prototypes represent parameterization of component type, component implementation, and feature group type declarations. They allow classifiers and features to be supplied when a component type, component implementation, or feature group is being extended or being used in a subcomponent declaration. The component classifier prototypes can be referenced in place of classifiers in feature declarations, in subcomponent declarations. The feature prototypes can be referenced in abstract feature declarations. Prototypes can also be referenced as actuals in prototype bindings; this allows parameterization via prototype to be propagated down the system hierarchy.

Syntax

```
prototype ::=
    defining_prototype_identifier :
      ( component_prototype
      | feature_group_type_prototype
      | feature_prototype )
      [ { { prototype_property_association }+ } ] ;

component_prototype ::=
    component_category [ unique_component_classifier_reference ] [ [] ]

feature_group_type_prototype ::=
    feature group [ unique_feature_group_type_reference ]

feature_prototype ::=
    [  in | out ] feature [unique_component_classifier_reference ]

prototype_refinement ::=
    defining_prototype_identifier : refined to
      ( component_prototype
      | feature_group_type_prototype
      | feature_prototype )
      [ { { prototype_property_association }+ } ];

prototype_bindings ::=
    ( prototype_binding { , prototype_ binding }* )

prototype_binding ::=
    prototype_identifier =>
      ( component_prototype_actual | component_prototype_actual_list
      | feature_group_type_prototype_actual | feature_prototype_actual )
```

```
component_prototype_actual ::=
    component_category
        ( unique_component_classifier_reference [ prototype_bindings ]
          | prototype_identifier )

component_prototype_actual_list ::=
    ( component_prototype_actual { , component_prototype_actual }* )

feature_group_type_prototype_actual ::=
      ( feature group unique_feature_group_type_reference )
      | (feature group feature_group_type_prototype_identifier )

feature_prototype_actual ::=
      (( in | out | in out) ( event | data | event data ) port ) |
      ( ( requires | provides )
          ( bus | data | subprogram group | subprogram ) access )
          unique_component_classifier_reference )
      | ( [  in | out ] feature feature_prototype_identifier )
```

Naming Rules

(N1) The prototype identifier on the left-hand side of a prototype binding must exist in the local namespace of the classifier for which the prototype binding is defined.

(N2) The prototype identifier on the right-hand side of a prototype binding, if present, must exist in the local namespace of the classifier that contains the prototype binding.

(N3) Unique component classifier references must exist in the public section of the package being identified in the reference.

(N4) Unique feature group type references must exist in the public section of the package being identified in the reference.

Legality Rules

(L1) The component category declared in the component prototype binding must match the component category of the prototype or classifier being referenced, i.e., they must be identical, or the declared category component category of the prototype must be **abstract**.

(L2) If the component prototype only specifies a component category, then any component type and component implementation of that category is acceptable; in the case of the category **abstract** any component type and component implementation of any category is acceptable.

(L3) The `Prototype_Substitution_Rule` property specifies the rules to be applied to determine an acceptable classifier supplied to the prototype. This property can be associated with a prototype declaration or the enclosing component type or component implementation. The rules are the same as those of the `Classifier_Substitution_Rule` property.

(L4) The category of the component implementation that contains the prototype declaration places restrictions on the set of acceptable categories for the prototype declaration and the supplied classifiers. The nesting rules for each category are defined in the respective component category section of this document. For example, if the component implementation is a **thread group** implementation, then the prototype referenced in a subcomponent declaration must be of the category **thread group**, **thread**, **subprogram, subprogram group**, **data**, or **abstract**.

(L5) In the case of feature prototypes, the supplied feature category must match the feature category in the prototype, or the feature category in the prototype must be **feature**.

(L6) If the direction is declared for feature prototypes, then the prototype actual satisfies the direction according to the same rules as for feature refinements (see Section 8); in the case of ports the direction must be **in** or **out**; in the case of data access, the access right must be read-only for **in** and write-only for **out**; in the case of bus access, subprogram access and subprogram group access the direction must not be **in** nor **out**.

(L7) In the case of feature group prototypes, the supplied feature group types must match the declared feature group type, if any. The `Prototype_Substitution_Rule` property rules apply to feature group types instead of component types.

(L8) A classifier supplied in a feature prototype binding must match the classifier of the prototype declaration, if present, according to the `Prototype_Substitution_Rule` property rules.

(L9) Component prototypes declared with square brackets specify that they expect a list of component classifiers. These prototypes can only be referenced in subcomponent array declarations. The component classifier list supplies the classifiers for each of the elements in the component array.

Standard Properties

```
Prototype_Substitution_Rule: inherit enumeration (Classifier_Match, Type_Extension,
Signature_Match)
```

Semantics

(2) Prototypes can specify a parameterization of component classifiers that can be referenced in feature declarations or in subcomponent declarations. The same prototype can be referenced several times in a component type and its component implementations to indicate that the same actually supplied classifier is to be used. The supplied component classifier may include prototype bindings if the classifier has unbound prototypes. Such a component classifier is effectively an unnamed extension of the classifier being referenced (see Section 4.5).

(3) Prototypes can specify a parameterization of abstract features (**feature**) as well as feature group types for feature groups. The prototype binding of an abstract feature can supply concrete features. If a direction is specified for the abstract feature, the direction of the supplied feature must match.

(4) Prototypes can only be bound once. Prototypes can be referenced in prototype bindings, i.e., bound classifiers and features can be passed down the component hierarchy.

(5) The prototype declaration specifies constraints on the component category, on the feature kind, and on the classifier that can be supplied. The `Prototype_Substitution_Rule` property specifies whether the match requires matching classifiers, allows classifier substitution, or allows any classifier with matching signature.

(6) A prototype refinement can increase the constraints on classifiers to be supplied. The newly specified category, classifier, and array dimensions must satisfy the same matching rules as the prototype bindings.

Examples

(7) This example defines a generic component with a flow through one in port and one out port. The abstract component type specifies the data type used on the port as one prototype and an incoming abstract feature as second prototype. This allows us to supply an event data or data port as the incoming port for this pattern. The outgoing port has been fixed to be a data port. The example also defines a primary/backup redundant implementation of the flow component as a pattern. It has a single prototype, namely the component that is to be implemented as a dual redundant component. This prototype is used to specify that both copies of the subcomponent are of the same classifier. It takes the data type prototype as its prototype actual to ensure that the data type of the pattern and the data type of the supplied control prototype will match. These abstract components are then refined into a controller process and its dual redundant instantiation as a system.

```
-- a generic component interface with one in and one out port
abstract flowComponent
prototypes
dt: data;
incoming: in feature;
features
 insignal: in feature incoming;
 outsignal: out data port dt;
end flowComponent;

-- a dual redundant component pattern
abstract implementation flowComponent.primaryBackup
prototypes
  control: abstract flowComponent;
subcomponents
  primary: abstract control;
  backup:abstract control;
connections
  inprimary: insignal -> primary.insignal;
  inbackup: insignal -> backup.insignal;
  outprimary: port primary.outsignal -> outsignal;
  outbackup: port backup.outsignal -> outsignal;
modes
  Primarymode: initial mode;
  Backupmode: mode;
end flowComponent.primaryBackup;

data signal
end signal;

data implementation signal.unit16
end signal.unit16;
```

```
-- a controller to be realized as dual redundant system
process controller extends flowComponent ( dt => data signal.unit16,
                              incoming => event data port signal.unit16 )
end controller;

-- the dual redundant controller system interface
system DualRedundantController extends
       flowComponent (dt => data signal.unit16,
                              incoming => event data port signal.unit16)
end DualRedundantController;

-- the dual redundant instance of the controller
system implementation DualRedundantController.PrimaryBackup
    extends flowComponent.primaryBackup (control => process controller)
end DualRedundantController.PrimaryBackup;
```

4.8 Annex Subclauses and Annex Libraries

(1) Annex subclauses allow annotations expressed in a sublanguage to be attached to component types, component implementations, and feature group types. Examples of standardized sublanguages are defined in the Error Model Annex Document C and in the Behavior Model Annex Document D.

(2) Annex libraries contain reusable declarations expressed in a sublanguage and are declared in packages. Those reusable declarations can be referenced by annex subclauses.

(3) A major use of these annex declarations is to accommodate new analysis methods through analysis specific notations or sublanguages.

(4) The AADL standard consists of a core language document and a set of standardized annex documents, some of which introduce sublanguage notations. Individual projects can introduce project-specific sublanguage notations to support project-specific analysis needs. Use of such sublanguage notations results in AADL models that are compliant with the core language standard. Standard compliant AADL tools are required to accept such AADL specifications (see *Processing Requirements and Permissions*).

(5) Examples of annex libraries are error states machines that can be associated with components in annex subclauses (see Annex Document C).

Syntax

```
annex_subclause ::=
    annex annex_identifier (
      ( {** annex_specific_language_constructs **} ) | none )
      [ in_modes ] ;

annex_library ::=
    annex annex_identifier
      (( {** annex_specific_reusable_constructs **} ) | none ) ;
```

Naming Rules

(N1) The annex identifier must be the name of an approved annex or a project-specific identifier different from the approved annex identifiers.

(N2) The mode identifiers in the `in_modes` statement must refer to modes in the component type or component implementation for which the annex subclause is declared.

Legality Rules

(L1) Annex subclauses can only be declared in component types, component implementations, and feature group types.

(L2) A component type, component implementation, or feature group type declaration may contain at most one annex subclause for each annex. If the annex subclause has an `in_modes` statement, then there must be at most one annex subclause per mode for each annex.

(L3) Annex libraries must be declared in packages.

(L4) A package declaration may contain at most one annex library declaration for each annex.

Semantics

(6) An annex subclause provides additional specification information about a component to be interpreted by analysis methods. Annex subclauses apply to component types and component implementations. Such annex subclauses can introduce analysis specific notations such as constraints and assertions expressed in predicate logic or behavioral descriptions expressed in temporal logic. Such notation can refer to subcomponents, connections, modes, and transitions as well as features and subcomponent access.

(7) Annex subclauses can be declared to be applicable to specific modes by specifying them with an `in_modes` statement. An annex subclause without an `in_modes` statement contains annex statements that are applicable in all modes. This capability allows users to attach mode specific annex annotations to core AADL models.

(8) An annex library provides reusable specifications expressed in an annex specific notation. Users can place multiple reusable annex specific constructs inside an annex library declaration. An example of a reusable annex specification is a fault state machine as defined by the Error Model Annex Document C.

Processing Requirements and Permissions

(9) Processing methods compliant with the core AADL standard must accept AADL specifications with approved and project-specific annex subclauses and specifications, but are not required to process the content of annex subclauses and annex library declarations. An AADL analysis tool must provide the option to report the use of project-specific annex sublanguages. Processing methods compliant with a given annex sublanguage must process specifications as defined in that annex sublanguage.

(10) Annex specific sublanguages can use any vocabulary word except for the symbol **}** representing the end of the annex subclause or specification.

(11) Annex specific sublanguages may introduce reserved words that may be the same or different from those in the core language or other annex sublanguages. If the annex sublanguage uses a reserved word that is a legal identifier in the AADL core language, then it must support the ability to refer to this named element in the core model.

(12) Annex specific sublanguages can utilize the core language property mechanism, i.e., properties can be defined in property sets that apply to elements in the sublanguage annex. For example, a property occurrence can be defined to apply to an error event in an error model.

(13) Annex sublanguages may choose not to support inheritance of sublanguage declarations contained in annex libraries of ancestor component type or component implementation declarations by their extensions.

Examples

```
thread Collect_Samples
features
   Input_Sample : in data port Sampling::Sample;
   Output_Average : out data port Sampling::Sample;
annex Error_Model {**
   Model => Transient_Fault_Model;
   Occurrence => 10e-4 poisson applies to Transient_Fault;
**};
end Collect_Samples;
```

5 Software Components

(1) This section defines the following categories of software components: data, subprogram, subprogram group, thread, thread group, and process.

(2) Software components may have associated source text specified using property associations. Software source text can be processed by source text tools to obtain a binary executable image consisting of code and data to be loaded onto a memory component and executed by a processor component. Source text may be written in a traditional programming language, a very-high-level or domain-specific language, or may be an intermediate product of processing such representations, e.g., an object file.

(3) Data components classifiers represent data types, while data subcomponents represent static data in source text, and local variables in subprograms. Data components are sharable between threads within the same thread group or process, and across processes and systems.

(4) The subprogram component models callable source text that is executed sequentially. Subprograms are callable from within threads and subprograms.

(5) Threads represent sequential sequences of instructions in loaded binary images produced from source text. Threads model schedulable units of control that can execute concurrently. Threads can interact with each other through exchanges of control and data as specified by port connections, through remote subprogram calls, and through shared data components.

(6) A thread group is a compositional component that permits organization of threads within processes into groups with relevant property associations.

(7) A process represents a virtual address space. Access protection of the virtual address space is by default enforced at runtime, but can be disabled if specified by the property `Runtime_Protection`. The source text associated with a process forms a complete program as defined in the applicable programming language standard. A complete process specification must contain at least one thread declaration. Processes may share a data component as specified by the required subcomponent resolved to an actual subcomponent and accessed through port connections.

5.1 Data

(1) A data component type represents a data type in source text. The internal structure of a source text data type, e.g., the instance variables of a class or the fields of a record, can be represented by data subcomponents in a data component implementation. The provides subprogram access features of a data component type can model the concept of methods on a class or operations on an abstract data type. A source text data type can be modeled by a data component type declaration with relevant properties without providing internal details in a data component implementation.

(2) A data component classifier, i.e., a data component type name or a data component type and implementation name pair (separated by a dot "."), is used as data type indicator in port declarations, subprogram parameter declarations, and data subcomponent declarations.

(3) A data subcomponent represents static data in the source text. Components can have shared access to data subcomponents, which means that there are mutual exclusion requirements. Only those components that explicitly declare required data access can access such sharable data subcomponents using a concurrency control protocol as specified by property. Data subcomponents can be shared within the same process and, if supported by the runtime system, across processes. When declared in a subprogram or a thread that data subcomponent represents a local variable.

(4) References to data components are modeled through provides and requires data access. Threads, processes, systems, and subprograms may access data by reference (see Section 8.6).

NOTES:

Support for shared data is not intended to be a substitute for data flow communication through ports. It is provided to support modeling of systems whose application logic requires them to manipulate data concurrently in a non-deterministic order that cannot be represented as data flow, such as database access. It is also provided for modeling source text that does not use port-based communication.

AADL focuses on architecture modeling. When used for data modeling, properties defined as part of the Data Modeling Annex (Annex Document B) can be used to further characterize data components.

Legality Rules

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

(L1) A data type declaration can contain provides subprogram access declarations as well as property associations.

(L2) A data type declaration must not contain a flow specification or modes subclause.

(L3) A data implementation can contain abstract, data and subprogram subcomponents, and data property associations.

(L4) A data implementation must not contain a flow implementation, an end-to-end flow specification, or a modes subclause.

Standard Properties

```
Source_Data_Size: Size
Source_Code_Size: Size
Type_Source_Name: aadlstring
Source_Name: aadlstring
Source_Text: inherit list of aadlstring
-- hardware mapping
Base_Address: aadlinteger 0 .. Max_Base_Address
Allowed_Memory_Binding_Class:
   inherit list of classifier (memory, system, processor)
Allowed_Memory_Binding: inherit list of reference (memory, system, processor)
Actual_Memory_Binding: inherit list of reference (memory)
-- Data sharing properties
Access_Right : Access_Rights => read_write
Concurrency_Control_Protocol: Supported_Concurrency_Control_Protocols
```

(5) The value of the `Type_Source_Name` property identifies the name of the data type declaration in the source text. The value of the `Source_Name` property identifies the name of the static or local data variable in the source text.

Semantics

(6) The data component type represents a data type in the source text that defines a representation and interpretation for instances of data in the source text. This includes data transferred through data and event data ports, and parameter values transferred to subprograms. This data type (class) may have associated access functions (called methods in an Object-Oriented context) that are represented by provides subprogram access declarations in the **features** subclause of the data type declaration. In this case, the data may be accessed through the subprograms.

(7) A provides subprogram access declaration represents a callable subprogram. It represents implicitly declared subprogram subcomponent that is contained in the process of the subprogram call, i.e., a single subprogram instance exists within a process. Modelers may also declare subprogram subcomponents explicitly. A remotely callable access function of a data type is modeled by a provides subprogram access declaration that will be connected from a subprogram call or is named in the classifier reference of the call.

(8) Elements of a data component can be accessed and changed directly or via provides subprogram access features. This corresponds to get and set methods of a class. AADL does not impose visibility restrictions on elements of a data component.

(9) A data component type can have zero data component implementations. This allows source text data types to be modeled without having to represent implementation details.

(10) A data component implementation represents the internal structure of a data component type. It can contain data subcomponents. This is used to model source language concepts such as fields in a record and instance variables in a class. The data subcomponent represents actual data values. AADL does not require the user to provide internal details of data representations if they are not relevant to the architecture model. The user may choose to reflect relevant data modeling information in properties, e.g., the memory requirements, the measurement unit used for the data, acceptable data types for a union type, dimensionality of an array structure, the super types in a type hierarchy, or the data type of a reference.

(11) Data component types can be extended through component type extension declarations. This permits modeling of subclasses and type inheritance in source text. However, it is recommended to use the capabilities of the Data Modeling Annex to represent data model characteristics in AADL (see Annex Document B).

(12) A data subcomponent represents a data instance, i.e., data in the source text that is potentially sharable between threads and persists across thread dispatches. A data subcomponent is considered to be static data with the exception of data subcomponents in subprograms, which represent local data. Each declared data subcomponent represents a separate instance of source text data.

(13) A data subcomponent declared in a subprogram represents local data. This data cannot be made accessible outside the subprogram through a provides data access declaration.

(14) When declaring data subcomponents, it is sufficient for the component classifier reference of data subcomponent declarations to only refer to the data component type. An implementation method can generate a system instance and perform memory usage analysis if a `Source_Data_Size` property value is specified in the data component type.

(15) Data subcomponents that are not declared in subprograms can be shared between threads. This is expressed by requires data access declarations in the component type declarations of subprograms, threads, thread groups, processes, and systems. The access is resolved to data subcomponents or provides data access declarations. Each required reference to shared data may have its own `Access_Right` property value. Its value must be consistent with the value of the `Access_Right` property of the data component or a provides data access. The `Access_Right` property value of `Read_Only` on a data component indicates that the component contains a constant value that does not change.

(16) Concurrent access to shared data is coordinated according to the concurrency control protocol specified by the `Concurrency_Control_Protocol` property value associated with the data component. A thread is considered to be in a critical region when it is accessing a shared data component. When a thread enters a critical region a `Get_Resource` operation is performed on the shared data component (see *Runtime Support* in Section 5.1.1). Upon exit from a critical region a `Release_Resource` operation is performed (see *Runtime Support* in Section 5.1.1). If multiple data components with concurrency control protocols are accessed by a thread, the AADL runtime system must ensure that the critical regions are nested, i.e., the `Get_Resource` and `Release_Resource` operations are pair-wise nested for each data component.

(17) Concurrent access to shared data may be coordinated through *provides subprogram (group) access*. In this case, the concurrency control protocol specifies how the execution of subprograms on the data component is coordinated.

(18) Data component classifier references are also used to specify the data type for data and event data ports as well as subprogram parameters. When ports are connected or when required data access and subprogram parameters are resolved, the data component classifier references representing the data types must be compatible. This means that the data type of an out port must be compatible with the data type of an in port, the data type of a provided data access declaration or a declared data component must be compatible with the data type of a required data component, and the data type of an actual parameter must be compatible with that of the formal parameter of a subprogram. Compatibility is determined by the `Classifier_Matching_Rule` property (see Section 9.2).

(19) Data components can be declared as arrays of data subcomponents. Same as for other subcomponent array declarations this is a short-hand for declaring several subcomponents of the same type through separate subcomponent declarations. If the intent is to model a data component whose representation in the source text is an array data modeling properties should be used (see Appendix A.8).

5.1.1 Runtime Support For Shared Data Access

(20) A standard set of runtime services is provided. The application program interface for these services is defined in the code generation annex of this standard (see Annex Document A). They are callable from within the source text.

(21) The following are subprograms that may be explicitly called by application source code, or they may be called by an AADL runtime system that is generated from an AADL model.

(22) The `Get_Resource` and `Release_Resource` runtime services represent an abstract interface for functions that perform locking and unlocking of resources according to the specified concurrency control protocol. The method may lock multiple resources simultaneously.

```
    subprogram  Get_Resource
    features
      resource: in parameter <implementation-specific representation of one or more
resources>;
    end Get_Resource;
    subprogram  Release_Resource
    features
      resource: in parameter <implementation-specific representation of one or more
resources>;
    end Release_Resource;
```

Processing Requirements and Permissions

(23) If any source text is associated with a data component type, then a corresponding source text data type declaration must be visible in the outermost scope of that source text, as defined by the scope and visibility rules of the applicable source language standard. The name of the data component type determines the source name of the data type. Supported mappings of the identifier to a source type name for specific source languages are defined in the source language annex of this standard. Such mapping can also be explicitly specified through the `Type_Source_Name` property.

(24) The applicable source language standard may allow a data type to be declared using a type constructor or type modifier that references other source text data types. A source text data type name defined by a source type constructor may, but is not required to, be modeled as a data component type with the referenced type features explicitly named in a corresponding data component implementation declaration.

(25) Data modeling properties allow for modeling data representations in the source text that include stored references (pointers). A method of implementation may disallow storing of such data references as data values in order to assure safe execution of embedded applications.

(26) A method of implementation may use a provides subprogram access declaration to represent an implicit subprogram subcomponent. In this case the provides subprogram access feature does not have to be connected to a subprogram subcomponent or a requires subprogram access feature.

(27) A method of implementation may disallow assignments that might result in a runtime error depending on the actual value being assigned. If a method of implementation employs a runtime check to determine if a specific value may be legally assigned, then any runtime fault is associated with the thread that contains the source of the data assignment.

(28) If two static data declarations refer to the same source text data, then that data must be replicated in binary images. If this replication occurs within the same virtual address space, a method for resolving name conflicts must be in place. Alternatively the processing method may require that each source text data be represented by only one data component declaration per process address space.

(29) The concurrency control protocol can be implemented through a number of concurrency control mechanisms such as mutex, lock, semaphore, or priority ceiling protocol. Appropriate concurrency control state is associated with the shared data component to maintain concurrency control. The protocol implementation must provide appropriate implementations of the `Get_Resource` and `Release_Resource` operations.

(30) A method of implementation may choose to support only locking of one resource at a time, or locking of multiple resources simultaneously. In the former case it is the responsibility of the caller of `Get_Resource` in such an order that deadlock and starvation is avoided. In the latter case, the `Get_Resource` implementation must assure absence of deadlock and starvation.

(31) A method of implementation may choose to generate the `Get_Resource` and `Release_Resource` calls as part of the AADL runtime system generation, or it may choose to require the application programmer to place those calls into the application code. In the latter case, implementation methods may validate the sequencing of those calls to assure compliance with the AADL specification.

Examples

```
package personnel
public
data Person
end Person;

data Personnel_record
-- Methods are not required, but when provided act as access methods
```

```
features
   -- a method on the data type
   -- subprogram type for signature
   update_address: provides subprogram access update_address;
end Personnel_record;

data implementation Personnel_record.others
subcomponents
   -- here we declare the internal structure of the data type
   -- One field is defined in terms of another type;
   -- the type name is sufficient, it defaults to others.
   -- the package Base_Types is defined in the Data Model Annex document.
   -- It provides data component classifiers for common data types.
   Name : data Base_Types::String;
   Home_address : data retep::relief::Address;
end Personnel_record.others;

data Personnel_database
end Personnel_database;

data implementation Personnel_database.oracle
end Personnel_database.oracle;

subprogram update_address
features
   person: in out parameter Personnel_record;
   street :in parameter Base_Types::String;
   city: in parameter Base_Types::String;
end update_address;
-- use of a data type as port type.
thread SEI_Personnel_addition
features
   new_person: in event data port Personnel_record;
   SEI_personnel: requires data access Personnel_database.oracle;
properties
   Dispatch_Protocol => aperiodic;
end SEI_Personnel_addition;
end personnel;
```

```
package retep::relief
public
   data Address
   features
      -- a subprogram access feature without parameter detail
      getStreet : provides subprogram access;
      getCity : provides subprogram access;
   end Address;
   data implementation Address.others
   properties
      Sourcee_Data_Size => 512 Bytes;
   end Address.others;
end retep::relief;

-- The implementation is shown as a private declaration
-- The public and the private part of a package are separate AADL specifications
package retep::relief
private
   data implementation Address.others
   subcomponents
      street : data Base_Types::String;
      streetnumber: data Base_Types::Integer;
      city: data Base_Types::String;
      zipcode: data Base_Types::Integer;
   end Address.others;
end retep::relief;
```


5.2 Subprograms and Subprogram Calls

(1) A subprogram component represents sequentially executed source text that is called with parameters. A subprogram may not have any state that persists beyond the call (static data). Subprograms can have local variables that are represented by data subcomponents in the subprogram implementation. All parameters and required access to external data must be explicitly declared as part of the subprogram type declaration. In addition, any events raised within a subprogram must be specified as part of its type declaration.

(2) A subprogram call sequence is declared in a thread implementation or in a subprogram implementation. Subprogram call sequences may be mode-specific. Subprograms can be called from threads and from other subprograms that execute within a thread. These calls can be local calls, i.e., performed in the context of the caller thread, or they can be remote calls to subprograms that are executed in the context of another thread.

(3) Subprogram instances may be modeled explicitly through subprogram subcomponent declarations. In this case, components can be modeled as requiring and providing access to subprogram instances.

(4) Subprogram instances may be implied by a subprogram call reference to a subprogram type or implementation. In this case, a subprogram is implicitly instantiated within the containing process.

Syntax

```
subprogram_call_sequence ::=
    defining_call_sequence_identifier :
    { { subprogram_call }+ }
        [ { { call_sequence_property_association }+ } ] [ in_modes ] ;

subprogram_call ::=
    defining_call_identifier : subprogram called_subprogram
       [ { { subcomponent_call_property_association }+ } ] ;


called_subprogram ::=
    -- identification by classifier
    subprogram_unique_component_classifier_reference
    | ( data_unique_component_type_reference
        . data_provides_subprogram_access_identifier )
    | ( subprogram_group_unique_component_type_reference
        . provides_subprogram_access_identifier )
    -- identification by prototype
    | prototype_identifier
    -- identification by processor subprogram access feature
    | ( processor . provides_subprogram_access_identifier )
    -- identification by subprogram instance
    | subprogram_subcomponent_identifier
    | ( subprogram_group_subcomponent_identifier .
        provides_subprogram_access_identifier )
    | requires_subprogram_access_identifier
    | ( requires_subprogram_group_access_reference .
        provides_subprogram_access_identifier )
```


NOTES:

Subprogram type and implementation declarations follow the syntax rules for component types and implementations. Subprogram instances may be implied by subprogram calls referring to subprogram classifiers, or subprogram instances may be declared explicitly as subprogram subcomponents and made accessible to calls through provides and requires subprogram access declarations.

Naming Rules

(N1) The defining identifier of a subprogram call sequence declaration must be unique within the local namespace of the component implementation that contains the subprogram call sequence.

(N2) The defining identifier of a subprogram call declaration must be unique within the local namespace of the component implementation that contains the subprogram call.

(N3) If the called subprogram name is a subprogram classifier reference, its component type identifier or component implementation name must exist in the package namespace.

(N4) The subprogram classifier reference of a subprogram call may be a subprogram type reference.

(N5) If the called subprogram name is a subprogram subcomponent reference, the subprogram subcomponent must exist in the component implementation containing the subprogram call declaration.

(N6) If the called subprogram name is a requires subprogram access reference, the requires subprogram access must exist in the component type of the component implementation containing the subprogram call declaration.

Legality Rules

Category	Type	Implementation
subprogram	Features: • out event port • out event data port • feature group • requires data access • requires subprogram access • requires subprogram group access • parameter • feature Flow specifications: yes Modes: yes Properties: yes	Subcomponents: • data • abstract Subprogram calls: yes Connections: yes Flows: yes Modes: yes Properties: yes

(L1) A subprogram type declaration can contain parameter, out event port, out event data port, and feature group declarations as well as requires data, subprogram, and subprogram group access declarations. It can also contain a flow specification subclause, a modes subclause, and property associations.

(L2) A subprogram implementation can contain abstract and data subcomponents, a subprogram calls subclause, a connections subclause, a flows subclause, a modes subclause, and property associations.

(L3) Only one subprogram call sequence can apply to a given mode.

Consistency Rules

(C1) The reference to a provides subprogram access of a processor in a subprogram call (**processor** . *provides_subprogram_access*_identifier) must identify a provides subprogram access feature of the processor that the thread executing the call is bound to.

(C2) A subprogram call may reference a subprogram classifier. A project may enforce a consistency rule that this reference be to a subprogram subcomponent declaration or requires subprogram access declaration. This ensures that a modeler either consistently uses explicit subprogram subcomponent declarations and references them.

Standard Properties

```
-- Properties related to source text
Source_Name: aadlstring
Source_Text: inherit list of aadlstring
Source_Language: inherit list of Supported_Source_Languages
Type_Source_Name: aadlstring
-- Properties specifying memory requirements of subprograms
Source_Code_Size: Size
Source_Data_Size: Size
Source_Stack_Size: Size
```

```
Source_Heap_Size: Size

Allowed_Memory_Binding_Class:
   inherit list of classifier (memory, system, processor)

Allowed_Memory_Binding: inherit list of reference (memory, system, processor)

Actual_Memory_Binding: inherit list of reference (memory)

 -- execution related properties

Compute_Execution_Time: Time_Range

Compute_Deadline: Time

Client_Subprogram_Execution_Time: Time_Range

Reference_Processor: inherit classifier ( processor )

-- remote subprogram call related properties

Urgency: aadlinteger 0 .. Max_Urgency

Actual_Subprogram_Call: reference (subprogram)

Allowed_Subprogram_Call: list of reference (subprogram)

Actual_Subprogram_Call_Binding: inherit list of reference (bus, processor, memory,
device)

Allowed_Subprogram_Call_Binding:
   inherit list of reference (bus, processor, device)

Subprogram_Call_Type: enumeration (Synchronous, SemiSynchronous)

    => Synchronous
```

Semantics

(5) A subprogram component represents sequentially executable source text that is called with parameters. The results of a subprogram call must be available to the caller at the time those results are used. This allows for synchronous and semi-synchronous calls.

(6) A subprogram type declaration specifies all interactions of the subprogram with other parts of the application source text. Subprogram parameters are specified as features of a subprogram type (see Section 8.4). This includes **in** and **in out** parameters passed into a subprogram and **out** and **in out** parameters returned from a subprogram on a call, events being raised from within the subprogram through its **out event port** and **out event data port**, required access to static data by the subprogram are specified as part of the features subclause of a subprogram type declaration, and required access to subprograms that are contained in another component and are called by this subprogram.

(7) A subprogram implementation represents implementation details that are relevant to architecture modeling. It specifies calls to other subprograms and the mode in which the call sequence occurs. It also specifies any local data of the subprogram, i.e., data that does not persist beyond the call.

(8) All access to data that persists beyond the life of the subprogram execution, i.e., any state that is maintained by a subprogram, must be modeled through requires data access. If requires data access is declared for a subprogram type, access to the data subcomponent may be performed in a critical region to assure concurrency control for calls from different threads (for more on concurrency control see Sections 5.1 and 5.4).

(9) Subprogram source text can contain `Send_Output` service calls to cause the transmission of events and event data through its **out event** ports (see Section 8.3). The fact that events may emit from a subprogram call is documented by the declaration of **out event ports** and **out event data ports** as features of the subprogram.

(10) Subprogram implementations and thread implementations can contain subprogram calls. A thread or subprogram can contain multiple calls to the same subprogram - with the same parameter values or with different parameter values.

(11) Subprogram call sequences can be declared to apply to specific modes. In this case a call sequence is only executed if one of the specified modes is the current mode.

(12) Modeling of subprograms is not required and the level of detail is not prescribed by the standard. Instead it is determined by the level of detail necessary for performing architecture analyses or code generation.

(13) In an object-oriented application methods are called on an object instance and the object instance is available within the method by the name *this*. In AADL a subprogram call can identify the subprogram being called by the provides subprogram access feature of a data component. In AADL, the data component must be explicitly passed into a subprogram as parameter (by value) or as requires data access (by reference). Requires data access may require concurrency control to ensure mutual exclusion.

(14) Ordering of subprogram calls is by default determined by the order of the subprogram call declarations. Annex-specific notations, e.g., the Behavior Annex, can be introduced to allow for other call order specifications, such as conditional calls and iterations.

(15) The flow of parameter values between subprogram calls as well as to and from ports of enclosing threads is specified through parameter connection declarations (see Section 9.3).

(16) Subprogram instances may be modeled explicitly through subprogram subcomponent declarations, or they may be implied from the call references to subprogram classifiers. A subprogram instance means that the subprogram executable binaries exist in the load image of the containing process. For subprograms, whose source text implementation is reentrant, it is assumed that a single instance of the subprogram binaries exist in the process address space. In the case of remote subprogram calls a proxy may be loaded for the calling thread and the actual subprogram is part of the load image of the process with the thread servicing the remote subprogram call.

(17) A subprogram subcomponent declaration explicitly represents a subprogram instance that resides in the protected address space of the containing process. Subprogram calls refer to the subprogram subcomponent or to requires subprogram access declarations. In case of a requires subprogram access the call is local to a subprogram instance in the containing process, or is remote to a subprogram instance in another process. Subprogram access connection declarations identify the subprogram instance to be called.

(18) The standard permits modeling of subprograms and subprogram calls without requiring the declaration of subprogram instances. In this case, subprogram calls may refer to subprogram classifiers and the source language processing system will determine the subprogram instance to be called. In the case of remote subprogram calls the target subprogram is identified by subprogram call properties. An `Allowed_Subprogram_Call` property, if present, identifies the remote subprogram(s) that are allowed to be used in a call binding. An `Actual_Subprogram_Call` property records the actual binding to a subprogram or provides subprogram access feature. Constraints on the buses and processors over which such calls can be routed can be specified with the `Allowed_Subprogram_Call_Bindings` property.

(19) The following control flow semantics apply to subprogram calls, when the call refers to:

- Subprogram classifier: execution by the calling thread
- Provides subprogram access of data type: execution by the calling thread
- Subprogram subcomponent in calling thread: execution by the calling thread
- Provides subprogram access feature of a data component: execution by calling thread
- Subprogram access to subprogram component in enclosing thread group, process, or system: execution by calling thread
- Subprogram access to subprogram component in another thread group, process, or system: execution by calling thread
- Provides subprogram access of another thread: execution by called thread
- Provides subprogram access feature of a device: execution inside device
- Subprogram access of a processor: execution inside the processor (operating system)
- Subprogram classifier and the call has a subprogram call binding property that refers to provides subprogram access in other thread: execution by called thread

(20) The results of a subprogram call must be available to the caller at the time those results are used. In the case of a local call the results are available when the call returns, i.e., the call is performed as a synchronous call. In the case of remote call, the caller thread is by default suspended until the execution of the subprogram completes (synchronous call). The caller thread may issue multiple concurrently executing subprogram calls and wait for their result when needed (semi-synchronous call). The `Subprogram_Call_Type` property indicates whether synchronous or semi-synchronous calls are desired.

(21) In the case of a remote call, the thread servicing the subprogram call assures that only one call at a time is serviced. In other words, it acts as a critical region for all calls to provides subprogram access features of a thread.

(22) Provides subprogram access features may be declared for processors or devices. In the case of processors they represent operating system services provided by the processor. In the case of a device, they represent services on the device that can be invoked by the application software.

Processing Requirements and Permissions

(23) The subprogram call order defines a default execution order for the subprogram calls. Alternate call orders can be modeled in an annex subclause introduced for that purpose.

(24) The legality rules require that call declarations either refer only to subprogram classifiers or to subprogram instances (subcomponents and provides/requires subprogram access). This rule can be relaxed to allow a mix of both if this is appropriate for the development process.

(25) An implementation method may support synchronous calls only or also semi-synchronous calls.

Examples

```
data matrix
end matrix;

data weather_forecast
end weather_forecast;

data date
end date;

subprogram Matrix_delta
features
   A: in parameter matrix;
   B: in parameter matrix;
   result: out parameter matrix;
end Matrix_delta;

subprogram Interpret_result
features
   A: in parameter matrix;
   result: out parameter weather_forecast;
end Interpret_result;
```

```
data weather_DB
features
   getCurrent: provides subprogram access getCurrent;
   getFuture: provides subprogram access getFuture;
end weather_DB;

subprogram getCurrent
features
   result: out parameter Matrix;
end getCurrent;

subprogram getFuture
-- a subprogram whose source text sends an event
-- the subprogram also has access to shared data
features
   date: in parameter date;
   result: out parameter Matrix;
   bad_date: out event port;
   wdb: requires data access weather_DB;
end getFuture;

thread Predict_Weather
features
   target_date: in event data port date;
   prediction: out event data port weather_forecast;
   past_date: out event port;
   weather_database: requires data access weather_DB;
end Predict_Weather;

thread implementation Predict_Weather.others
calls {
   -- subprogram call on a data component provides subprogram access feature
   -- out parameter is not resolved, but will be identified by user of value
   current: subprogram weather_DB.getCurrent;

   -- subprogram call on a data component provides subprogram access feature with port
value
   -- as additional parameter. Event is mapped to thread event
   future: subprogram weather_DB.getFuture;
```

```
   -- in parameter actuals are out parameter values of previous calls
   -- they are identified by the call name and the out parameter name
   diff: subprogram Matrix_delta;

   -- call with out parameter value resolved to be passed on through a port
   interpret: subprogram Interpret_result;
   };
connections
   parameter target_date -> future.date;
   port future.bad_date -> past_date;
   parameter current.result -> diff.A;
   parameter future.result -> diff.B;
   parameter diff.result -> interpret.A;
   parameter interpret.result -> prediction;
   data access weather_database <-> future.wdb;
end Predict_Weather.others;
```


5.3 Subprogram Groups and Subprogram Group Types

(1) Subprogram groups represent subprogram libraries. Subprogram groups can be made accessible to other components through subprogram group access features (see Section 8.4) and subprogram group access connections (see Section 9.4). This grouping concept allows the number of connection declarations to be reduced, especially at higher levels of a system when a number of provided subprograms from one subcomponent and its contained subcomponents must be connected to requires subprogram access in another subcomponent and its contained subcomponents. The content of a subprogram group is declared through a subprogram group type declaration. This declaration is then referenced when subprogram groups are declared as subcomponents.

Naming Rules

(N1) The defining identifier of a subprogram group type must be unique within the package namespace of the package where the subprogram group type is declared.

(N2) Each subprogram group provides a local namespace. The defining subprogram identifiers of subprogram declarations in a subprogram group type must be unique within the namespace of the subprogram group type.

(N3) The local namespace of a subprogram group type extension includes the defining identifiers in the local namespace of the subprogram group type being extended. This means, the defining identifiers of subprogram or subprogram group declarations in a subprogram group type extension must not exist in the local namespace of the subprogram group type being extended. The defining identifiers of subprogram or subprogram group refinements in a subprogram group type extension must refer to a subprogram or subprogram group in the local namespace of an ancestor subprogram group type.

(N4) The defining subprogram identifiers of subprogram access feature declarations in feature group refinements must not exist in the local namespace of any subprogram group being extended. The defining subprogram identifier of `subprogram_refinement` declarations in subprogram group refinements must exist in the local namespace of any feature group being extended.

(N5) The package name of the unique subprogram group type reference must refer to a package name in the global namespace. The subprogram group type identifier of the unique subprogram group type reference must refer to a subprogram group type identifier in the named package.

Legality Rules

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

(L1) A subprogram group type can contain provides and requires subprogram access, and requires subprogram group access.

(L2) A subprogram group implementation can contain abstract and subprogram subcomponents and subprogram access connections.

(L3) A subprogram group type or implementation may contain zero or more subprograms. If it contains zero elements, then the subprogram group type or implementation is considered to be incompletely specified.

Standard Properties

```
-- Port properties defined to be inherit, thus can be associated with a
-- feature group to apply to all contained ports.
Source_Text: inherit list of aadlstring
-- properties related to execution time
Reference_Processor: inherit classifier ( processor )
-- Properties specifying memory requirements of subprograms
Source_Code_Size: Size
Source_Data_Size: Size
Source_Stack_Size: Size
Source_Heap_Size: Size
Allowed_Memory_Binding_Class:
   inherit list of classifier (memory, system, processor)
Allowed_Memory_Binding: inherit list of reference (memory, system, processor)
```

Semantics

(2) A subprogram group declaration represents groups of component subprograms, i.e., subprogram libraries. Subprograms in a subprogram group may require access to other subprograms or subprogram groups.

(3) Requires subprogram group access is resolved to provides subprogram group access or a subprogram group subcomponent.

(4) The subprograms of a subprogram group or a subprogram group access feature can be connected to or referenced in a subprogram call.

Processing Requirements and Permissions

(5) Subprogram groups represent subprogram libraries. These can be application libraries or system libraries. Libraries may be shared across multiple applications, i.e., across multiple processes.

(6) Methods of implementation may optionally allow a provides subprogram access declaration of a subprogram group to not be connected to a subprogram instantiation, i.e., subprogram subcomponent. It may assume these subprograms to be implicitly declared and instantiated as part of a subprogram group instantiation.

Examples

```
subprogram group mathlib
features
   matrixMultiply: provides subprogram access ;
   matrixAdd: provides subprogram access ;
   vectorAdd: requires subprogram access ;
end mathlib;
```

5.4 Threads

(1) A thread models a concurrent task or an active object, i.e., a schedulable unit that can execute concurrently with other threads. Each thread represents a sequential flow of control that executes instructions within a binary image produced from source text. One or more AADL threads may be implemented in a single operating system thread. A thread always executes within the virtual address space of a process, i.e., the binary images making up the virtual address space must be loaded before any thread can execute in that virtual address space. Threads are dispatched, i.e., their execution is initiated periodically by the clock or by the arrival of data or events on ports, or by arrival of subprogram calls from other threads.

(2) AADL supports an input-compute-output model of communication and execution for threads and port-based communication. The inputs received from other components are frozen at a specified point, by default the dispatch of a thread. As a result the computation performed by a thread is not affected by the arrival of new input until an explicit request for input, by default the next dispatch. Similarly, the output is made available to other components at a specified point in time, for data ports by default at completion time or thread deadline. In other words, AADL is able to support both synchronous execution and communication behavior, e.g., in the form of deterministic sampling of a control system data stream, as well as asynchronous concurrent processing.

(3) Systems modeled in AADL can have operational modes (see Section 12). A thread can be active in a particular mode and inactive in another mode. As a result a thread may transition between an active and inactive state as part of a mode switch. Only active threads can be dispatched and scheduled for execution. Threads can be dispatched periodically or as the result of explicitly modeled events that arrive at event ports, event data ports. Completion of the normal execution including error recovery will result in an event being delivered through the predeclared `Complete` event out port if it is connected.

(4) If the 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 thread unrecoverable, its occurrence is reported through the predeclared `Error` out event data port.

Legality Rules

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

(L1) A thread type declaration can contain port, feature group, requires and provides data access declarations, as well as requires and provides subprogram access declarations. It can also contain flow specifications, a modes subclause, and property associations.

(L2) A thread component implementation can contain abstract, data, subprogram, and subprogram group subcomponent declarations, a calls subclause, a flows subclause, a modes subclause, and thread property associations.

(L3) The `Complete` **out** event port, and `Error` **out** event data port are predeclared, i.e., are implicitly identifiers in the name space of a thread type. Therefore, there cannot be user-defined features with those names in threads.

Consistency Rules

(C3) Either the `Compute_Entrypoint`, `Compute_Entrypoint_Source_Text` or `Compute_Entrypoint_Call_Sequence` property must have a value that indicates the source code to execute after a thread has been dispatched when an implementation is to be generated or consistency with source code is to be checked. Other entrypoint properties are optional, i.e., if a property value is not defined, then the entrypoint is not called.

(C4) The Period property must have a value if the `Dispatch_Protocol` property value is periodic, sporadic, timed, or hybrid.

Standard Properties

```
-- Properties related to source text
Source_Text: inherit list of aadlstring
Source_Language: inherit list of Supported_Source_Languages
-- Properties specifying memory requirements of threads
Source_Code_Size: Size
Source_Data_Size: Size
Source_Stack_Size: Size
Source_Heap_Size: Size
-- Properties specifying thread dispatch properties
Dispatch_Protocol: Supported_Dispatch_Protocols
Dispatch_Trigger: list of reference (port)
```

```
Dispatch_Offset: inherit Time

First_Dispatch_Time : inherit Time

Input_Time: list of IO_Time_Spec => ( Time => Dispatch; Offset => 0.0 ns .. 0.0 ns;)

Input_Rate: Rate_Spec => ( Value_Range => 1.0 .. 1.0; Rate_Unit => PerDispatch;
Rate_Distribution => Fixed; )

Output_Time: list of IO_Time_Spec => ( Time => Completion; Offset => 0.0 ns .. 0.0 ns;)

Output_Rate: Rate_Spec => ( Value_Range => 1.0 .. 1.0; Rate_Unit => PerDispatch;
Rate_Distribution => Fixed; )

Period: inherit Time

-- the default value of the deadline is that of the period

Deadline: inherit Time => Period

-- Scheduling properties

Priority: inherit aadlinteger

POSIX_Scheduling_Policy : enumeration (SCHED_FIFO, SCHED_RR, SCHED_OTHERS)

Criticality: aadlinteger

Time_Slot: list of aadlinteger

 -- Properties specifying execution entrypoints and timing constraints

Initialize_Execution_Time: Time_Range

Initialize_Deadline: Time

Initialize_Entrypoint: classifier ( subprogram classifier )

Initialize_Entrypoint_Call_Sequence: reference ( subprogram call sequence )

Initialize_Entrypoint_Source_Text: aadlstring

Compute_Execution_Time: Time_Range

Compute_Deadline: Time

Compute_Entrypoint: classifier ( subprogram classifier )

Compute_Entrypoint_Call_Sequence: reference ( subprogram call sequence )

Compute_Entrypoint_Source_Text: aadlstring

Activate_Execution_Time: Time_Range

Activate_Deadline: Time

Activate_Entrypoint: classifier ( subprogram classifier )

Activate_Entrypoint_Call_Sequence: reference ( subprogram call sequence )

Activate_Entrypoint_Source_Text: aadlstring

Deactivate_Execution_Time: Time_Range

Deactivate_Deadline: Time

Deactivate_Entrypoint: classifier ( subprogram classifier )

Deactivate_Entrypoint_Call_Sequence: reference ( subprogram call sequence )

Deactivate_Entrypoint_Source_Text: aadlstring

Recover_Execution_Time: Time_Range

Recover_Deadline: Time

Recover_Entrypoint: classifier ( subprogram classifier )
```

```
Recover_Entrypoint_Call_Sequence: reference ( subprogram call sequence )
Recover_Entrypoint_Source_Text: aadlstring
Finalize_Execution_Time: Time_Range
Finalize_Deadline: Time
Finalize_Entrypoint: classifier ( subprogram classifier )
Finalize_Entrypoint_Call_Sequence: reference ( subprogram call sequence )
Finalize_Entrypoint_Source_Text: aadlstring

Reference_Processor: inherit classifier ( processor )
-- mode to enter as result of activation
Resumption_Policy: enumeration ( restart, resume )
-- Properties specifying constraints for processor and memory binding
Allowed_Processor_Binding_Class:
   inherit list of classifier (processor, virtual processor, system)
Allowed_Processor_Binding: inherit list of reference (processor, virtual processor,
system)

Allowed_Memory_Binding_Class:
   inherit list of classifier (memory, system, processor)
Allowed_Connection_Binding_Class:
   inherit list of classifier(processor, virtual processor, bus, virtual bus, device,
memory)
Allowed_Connection_Binding: inherit list of reference (processor, virtual processor, bus,
virtual bus, device, memory)

Not_Collocated: record (
    Targets: list of reference (data, thread, process, system, connection);
    Location: classifier ( processor, memory, bus, system ); )
Collocated: record (
    Targets: list of reference (data, thread, process, system, connection);
    Location: classifier ( processor, memory, bus, system ); )
-- Binding value filled in by binding tool
Actual_Processor_Binding: inherit list of reference (processor, virtual processor)
Actual_Memory_Binding: inherit list of reference (memory)
Actual_Connection_Binding: inherit list of reference (processor, virtual processor, bus,
virtual bus, device, memory)

-- property indicating whether the thread affects the hyperperiod
-- for mode switching
Synchronized_Component: inherit aadlboolean => true
-- property specifying the action for executing thread at actual mode switch
Active_Thread_Handling_Protocol:
   inherit Supported_Active_Thread_Handling_Protocols => abort
```

```
Active_Thread_Queue_Handling_Protocol:
   inherit enumeration (flush, hold) => flush
```

NOTE: Entrypoints for thread execution can be specified in three ways: by identifying source text name, by identifying a subprogram classifier representing the source text, or by a call sequence.

Compute_Entrypoint => **classifier** (ControlAlgorithm.basic);

Compute_Entrypoint_Call_Sequence => **reference** (callseq1);

Compute_Entrypoint_Source_Text => “MyControlAlgorithm”;

Semantics

(5) Thread semantics are described in terms of thread states, thread dispatching, thread scheduling and execution, and fault handling. Thread execution semantics apply once the appropriate binary images have been loaded into the respective virtual address space (see Section 5.6).

(6) Threads are dispatched periodically or by the arrival of data and events, or by arrival of subprogram calls from other threads. Subprogram calls always trigger dispatches. Subsets of ports can be specified to trigger dispatches. By default, any one of the incoming **event ports** and **event data ports** triggers a dispatch. The `Dispatch_Trigger` property can specify different subsets of ports, including **data ports** (see Section 5.4.2) as a disjunction or the Behavior Annex (see Section Annex Document D) can be used to specify additional logical conditions on thread dispatch triggering.

(7) Port input is frozen at dispatch time or a specified time during thread execution and made available to the thread for access in the form of a port variable (see Section 8.3). From that point on its content is not affected by new arrival of data and event for the remainder of the current execution. This assures a input-compute-output model of execution. By default, input of ports is frozen for all ports that are not candidates for thread dispatch triggering; for dispatch trigger candidates, only those port(s) actually triggering a specific dispatch is frozen. Whether input of specific ports is frozen at a dispatch and the time at which it is frozen can be explicitly specified (see Section 8.3.2).

(8) Threads may be part of modes of containing components. In that case a thread is active, i.e., eligible for dispatch and scheduling, only if the thread is part of the current mode (see Sections 5.4.1 and 13.6).

(9) Threads can contain mode subclauses that define thread-internal operational modes. Threads can have property values that are different for different thread-internal modes (see Section 5.4.5).

(10) Every thread has a predeclared **out event port** named `Complete`. If this port is connected, i.e., named as the source in a connection declaration, then an event is raised implicitly on this port when nominal execution including recovery of a thread dispatch completes.

(11) Every thread has a predeclared **out event data port** named `Error`. If this port is connected, i.e., named as the source in a connection declaration, then an event is raised implicitly on this port when a thread unrecoverable error is detected (see Section 5.4.4 for more detail). This supports the propagation of thread unrecoverable errors as event data for fault handling by a thread.

(12) Threads may contain subprogram subcomponents that can be called from within the thread, and also by other threads if it is made accessible through a provides subprogram access declaration. Similarly, a thread can contain a subprogram group declaration, which represents an instance of a subprogram library dedicated to the thread. The subprograms within the subprogram library can be called from within the thread or by other threads if it is made accessible through a provides subprogram group access declaration. For further details about calling subprograms see Section 5.2. Finally, a thread can contain data subcomponents. They represent static data owned by the thread, i.e., state that is preserved between thread dispatches. The thread has exclusive access to this data component unless it specifies it to be accessible through a provides data access declaration.

5.4.1 Thread States and Actions

(13) A thread executes a code sequence in the associated source text when dispatched and scheduled to execute. This code sequence is part of a binary image accessible in the virtual address space of the containing process. It is assumed that the process is bound to the memory that contains the binary image (see Section 5.6).

(14) A thread goes through several states. Thread state transitions under normal operation are described here and illustrated in Figure 5. Thread state transitions under fault conditions are described in Section 5.4.4

(15) The initial state is *thread halted*. When the loading of the virtual address space as declared by the enclosing process completes (see Section 5.6), a thread is *initialized* by performing an initialization code sequence in the source text. Once initialization is completed the thread enters the *suspended awaiting dispatch* state if the thread is part of the initial mode, otherwise it enters the *suspended awaiting mode* state. When a thread is in the *suspended awaiting mode* state it cannot be dispatched for execution.

(16) A thread may be declared to have modes. In this case, each mode represents a behavioral state within the execution of the thread. When a thread is dispatched it is assumed to execute in a specific mode. It may resume execution in the behavioral state (mode) in which it completed its previous dispatch execution, e.g., state reflected in a static data component, or it may execute in a specific mode based on the input received at dispatch time. A modal thread can have mode-specific property values. For example, a thread can have different worst-case execution times for different modes, each representing a different execution path through the source code. The result is a model that more accurately reflects the actual system behavior. The Behavior Model Annex Document D allows for a refined specification of thread behavior, e.g., it may explicitly specify the conditions under which the thread executes in one mode or another mode and it can represent intermediate behavioral states.

(17) When a mode transition is initiated, a thread that is part of the old mode and not part of the new mode *exits* the mode by transitioning to the *suspended awaiting mode* state after performing *thread deactivation* during the *mode change in progress* system state (see Figure 23). If the thread is periodic and its `Synchronized_Component` property is true, then its period is taken into consideration to determine the actual mode transition time (see Sections 12 and 13.6 for detailed timing semantics of a mode transition). If an aperiodic or a sporadic thread is executing a dispatch when the mode transition is initiated, its execution is handled according to the `Active_Thread_Handling_Protocol` property. The execution of a background thread is suspended through deactivation while the thread is not part of the new mode. A thread that is not part of the old mode and part of the new mode *enters* the *mode* by transitioning to the *suspended awaiting dispatch* state after performing *thread activation*.

(18) When in the *suspended awaiting dispatch* state, a thread is awaiting a dispatch request for performing the execution of a compute source text code sequence as specified by the `Compute_Entrypoint` property on the thread or on the event or event data port that triggers the dispatch. When a dispatch request is received for a thread, data, event information, and event data is made available to the thread through its port variables (see Sections 8.2 and 9.1). The thread is then handed to the scheduler to perform the computation. Upon successful completion of the computation, the thread returns to the *suspended awaiting dispatch* state. If a dispatch request is received for a thread while the thread is in the compute state, this dispatch request is handled according to the specified `Overflow_Handling_Protocol` for the event or event data port of the thread.

(19) A thread may enter the *thread halted* state, i.e., will not be available for future dispatches and will not be included in future mode switches. If re-initialization is requested for a thread in the *thread halted* state (see Section 5.6), then its virtual address space is reloaded, the processor to which the thread is bound is restarted, or the system instance is restarted.

(20) A thread may be requested to enter its *thread halted* state through a *stop* request after completing the execution of a dispatch or while not part of the active mode. In this case, the thread may execute a *finalize* entrypoint before entering the *thread halted* state. A thread may also enter the *thread halted* state immediately through an *abort* request. Any resources locked by `Get_Resource` are released (see Figure 5).

(21) Figure 5 presents the top-level hybrid automaton (using the notation defined in Section 1.6) to describe the dynamic semantics of a thread from the perspective of a scheduler. The hybrid automaton states complement the application modes declared for threads. Figure 7 elaborates the performing *thread computation* state of Figure 5. Figure 6 elaborates the executing *nominally* substate of Figure 7. The bold faced edge labels in Figure 5 indicate that the transitions marked by the label are coordinated across multiple hybrid automata. The scope of the labels is indicated in parentheses, i.e., interaction with the process hybrid automaton (Figure 8), with the system hybrid automaton (Figure 22) and with system wide mode switching (see Figure 23). Thread initialization is only started when the process containing the thread has been loaded as indicated by the label **loaded(process)**. The label **started(system)** is coordinated with other threads and the system hybrid automaton to transition to *System operational* only after threads have been initialized. In some systems it is desirable to initialize all threads, while in other system it is acceptable for threads to be created and initialized more dynamically, possibly even at activation and deactivation.

(22) The hybrid automata contain assertions. In a time-partitioned system these assertions will be satisfied. In other systems they will be treated as anomalous behavior.

(23) For each of the states representing a *performing thread* action such as *initialize*, *compute*, *recover*, *activate*, *deactivate*, and *finalize*, an execution entrypoint to a code sequence in the source text can be specified. Each entrypoint may refer to a different source text code sequence which contains the entrypoint, or all entrypoints of a thread may be contained in the same source text code sequence. In the latter case, the source text code sequence can determine the context of the execution through a `Dispatch_Status` runtime service call (see Section 5.4.8). The execution semantics for these entrypoints is described in Section 5.4.3.

(24) An *Initialize_Entrypoint* (enter the state `performing thread initialization` in Figure 5) is executed during system initialization and allows threads to perform application specific initialization, such as ensuring the correct initial value of its **out** and **in out** ports. A thread that has halted may be re-initialized.

(25) The *Activate_Entrypoint* (enter the state `performing thread activation` in Figure 5) and *Deactivate_Entrypoint* (enter the state `performing thread activation` in Figure 5) are executed during mode transitions and allow threads to take user-specified actions to save and restore application state for continued execution between mode switches. These entrypoints may be used to reinitialize application state due to a mode transition. Activate entrypoints can also ensure that **out** and **in out** ports contain correct values for operation in the new mode.

(26) The *Compute_Entrypoint* (enter state `performing thread computation` in Figure 5) represents the code sequence to be executed on every thread dispatch. Each provides subprogram access feature represents a separate compute entrypoint of the thread. Remote subprogram calls are thread dispatches to the respective entrypoint. Event ports and event data ports can have port specific compute entrypoints to be executed when the corresponding event or event data dispatches a thread.

(27) A *Recover_Entrypoint* (enter the state `executing recovery` in Figure 7) is executed when a fault in the execution of a thread requires recovery activity to continue execution. This entrypoint allows the thread to perform fault recovery actions (for a detailed description see Section 5.4.4).

(28) A *Finalize_Entrypoint* (enter the state `performing thread finalize` in Figure 5) is executed when a thread is asked to terminate as part of a process unload or process stop.

(29) If no value is specified for any of the entrypoints, then there is no invocation at all.

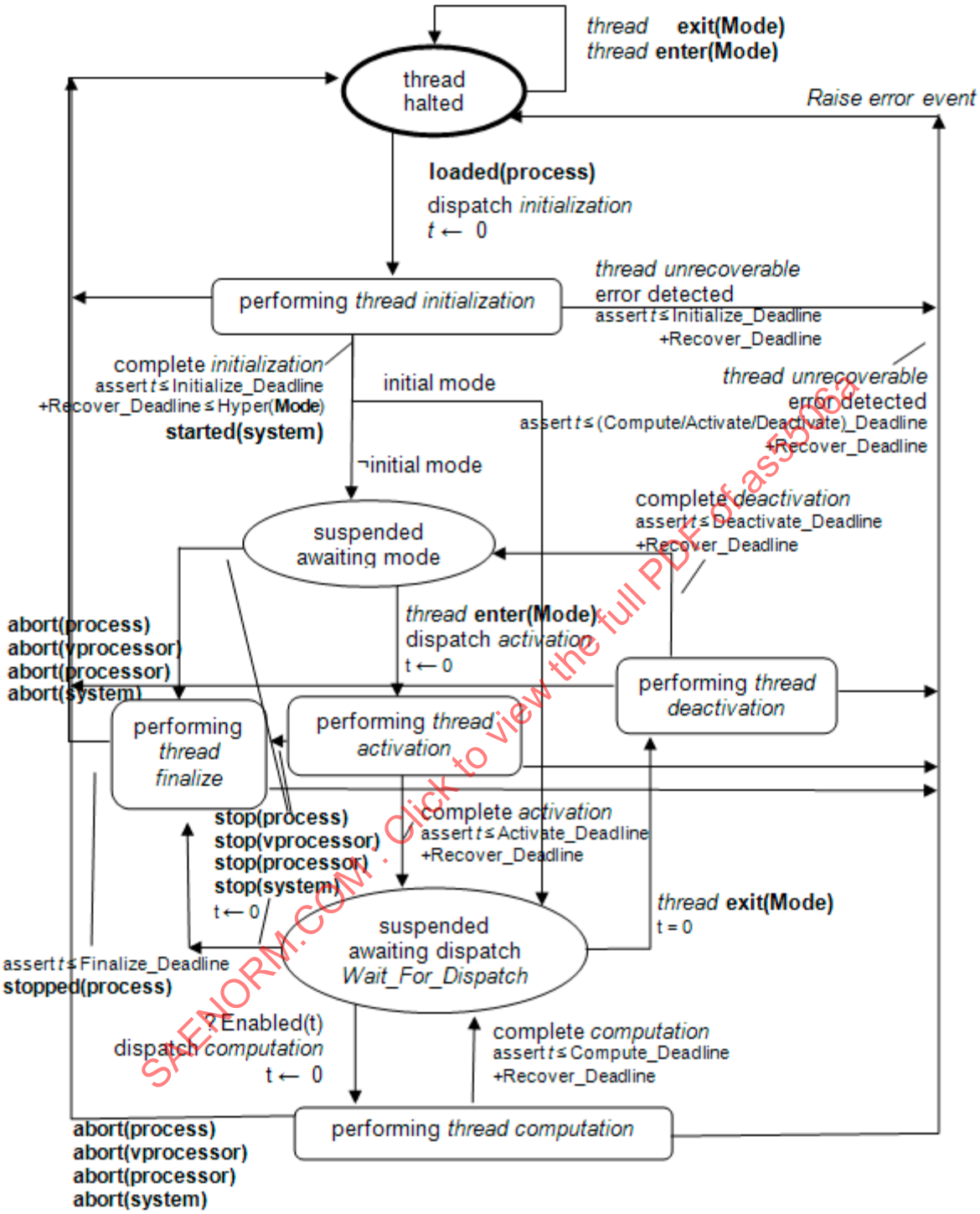


Figure 5 Thread States and Actions

5.4.2 Thread Dispatching

(30) Threads are dispatched periodically determined by a clock or by the arrival of events, event data, of calls to provides subprogram access. By default any event port or event data port can trigger a dispatch. In that case, only the input of the port triggering the dispatch and any data port is available to the application program.

(31) A thread may have a `Dispatch_Trigger` property to specify a subset of event, data, or event data ports that can trigger a thread dispatch. In this case, arrival of events or event data on any of the listed ports can trigger the dispatch.

(32) The default disjunction of ports triggering a dispatch can be overwritten by a logical condition on the ports expressed by a annex subclauses of the Behavior Annex notation (see Annex Document D).

(33) For periodic threads arrival of events or event data will not result in a dispatch. Events and event data are queued in their incoming port and are accessible to the application code of the thread. Periodic thread dispatches are solely determined by the clock according to the time interval specified through the `Period` property value.

(34) The `Dispatch_Protocol` property of a thread determines the characteristics of dispatch requests to the thread. This is modeled in the hybrid automaton in Figure 5 by the `Enabled(t)` function as the `Wait_For_Dispatch` invariant. The `Enabled` function determines when a transition from `Wait_For_Dispatch` to performing thread computation will occur. The `Wait_For_Dispatch` invariant captures the condition under which the `Enabled` function is evaluated. The consequence of a dispatch is the execution of the entrypoint source text code sequence at its *current execution* position. This position is set to the first step in the code sequence and reset upon completion (see Section 5.4.3).

(35) For a thread whose dispatch protocol is `periodic`, a dispatch request is issued at time intervals of the specified `Period` property value. The `Enabled` function is $t = Period$. The `Wait_For_Dispatch` invariant is $t \leq Period \wedge \delta t = 1$. The dispatch occurs at $t = Period$. The `Compute_Entrypoint` of the thread is called.

(36) Periodic threads can have a `Dispatch_Offset` property value. In this case the dispatch time is offset from the period by the specified amount. This allows two periodic threads with the same period to be aligned, where the first thread has a pre-period deadline, and the second thread has a dispatch offset greater than the deadline of the first thread. This is a static alignment of thread execution order within a frame, while the immediate data connection achieves the same by dynamically aligning completion time of the first thread and the start of execution of the second thread (see Section 9.2.5).

(37) For threads whose dispatch protocol is `aperiodic`, `sporadic`, `timed`, or `hybrid`, a dispatch request is the result of an event or event data arriving at an event or event data port of the thread, or a remote subprogram call arriving at a provides subprogram access feature of the thread. This *dispatch trigger condition* is determined as follows:

- Arrival of an event or event data on any incoming event, or event data port, or arrival of any subprogram call request on a provides subprogram access feature. In other words, it is a disjunction of all incoming features.
- By arrival on a subset of incoming features (port, subprogram access). This subset can be specified through the `Dispatch_Trigger` value for the `Input_Time` property of a feature.
- By a user-defined logical condition on the incoming features that can trigger the dispatch expressed through an annex subclause expressed in the Behavior Annex sublanguage notation (see Annex Document D).

(38) For a thread whose dispatch protocol is `aperiodic`, a dispatch request is the result of an event or event data arriving at an event or event data port of the thread, or a remote subprogram call arriving at a provides subprogram access feature of the thread. There is no constraint on the inter-arrival time of events, event data or remote subprogram calls. The dispatch actually occurs immediately when a dispatch request arrives in the form of an event at an event port with an empty queue, or if an event is already queued when a dispatch execution completes, or a remote subprogram call arrives. The `Enabled` function by default has the value true if there exists a port or provides subprogram access (`p`) in the set of features that can trigger a dispatch (E) with a non-empty queue, i.e., $\exists$ `p in E:` $p \neq \emptyset$. This evaluation function may be redefined by the Behavior Annex (see Annex Document D). The `Wait_For_Dispatch` invariant is that no event, event data, or subprogram call is queued, i.e., $\forall$ `p in E:` $p = \emptyset$. The `Compute_Entrypoint` of the port triggering the dispatch, or if not present that of the thread, is called.

(39) If multiple ports are involved in triggering the dispatch the `Compute_Entrypoint` of the thread is called. The list of ports actually satisfying the dispatch trigger condition that results in the dispatch is available to the source text as output parameter of the `Await_Dispatch` service call (see Section 5.4.8).

(40) For a thread whose dispatch protocol is `sporadic`, a dispatch request is the result of an event or event data arriving at an event or event data port of the thread, or a remote subprogram call arriving at a provides subprogram access feature of the thread. The time interval between successive dispatch requests will never be less than the associated `Period` property value. The `Enabled` function is `t ≥ Period∧∃p in E: p ≠ ∅`. The `Wait_For_Dispatch` invariant is `t < Period ∨ (t > Period ∧ ∀ p in E: p = ∅)`. The dispatch actually occurs when the time condition on the dispatch transition is true and a dispatch request arrives in the form of an event at an event port with an empty queue, or an event is already queued when the time condition becomes true, or a remote subprogram call arrives when the time condition is true. The `Compute_Entrypoint` of the port triggering the dispatch, or if not present that of the thread, is called.

(41) For a thread whose dispatch protocol is `timed`, a dispatch request is the result of an event, event data, or remote subprogram arrival, or it occurs by an amount of time specified by the `Period` property since the last dispatch.. In other words, the `Period` represents a time-out value that ensure a dispatch occurs after a given amount of time if no events, event data, or remote subprogram calls have arrived or are queued. The `Enabled` function by default has the value true if there exists a port or provides subprogram access (`p`) in the set of features that can trigger a dispatch (E) with an event, event data, or call in its queue, or time equal to the `Period` has expired since the last dispatch, i.e., `∃ p in E: p ≠∅ ∨ t = Period`. `t` is reset to zero at each dispatch. This evaluation function may be redefined by the Behavior Annex (see Annex Document D). The `Wait_For_Dispatch` invariant is that no event, event data, or call is queued, i.e., `∀p in E: p = ∅ ∧ t < Period`. The `Compute_Entrypoint` of the port triggering the dispatch, or if not present that of the thread, is called. If a timeout occurs, i.e., the dispatch is triggered at the end of the period, the `Recover_Entrypoint` is called.

(42) A thread whose dispatch protocol is `hybrid`, combines both `aperiodic` and `periodic` dispatch behavior in the same thread. A dispatch request is the result of an event, event data, or remote subprogram call arrival, as well as periodic dispatch requests at a time interval specified by the `Period` property value. The `Enabled` function is `t = Period ∨∃p in E: p ≠ ∅. t` is reset to zero at each periodic dispatch. The evaluation function for events, event data, or subprogram calls may be redefined by the Behavior Annex. The `Wait_For_Dispatch` invariant is that no event, event data, call, or periodic dispatch is queued and the period has not expired, i.e., `∀p in E: p = ∅ ∧ t ≤ Period`. The `Compute_Entrypoint` of the port triggering the dispatch, or if not present that of the thread, is called.

(43) If several events or event data occur logically simultaneously and are routed to the same port of an `aperiodic`, `sporadic`, `timed`, or `hybrid` thread, the order of arrival for the purpose of event handling according the above rules is implementation-dependent. If several events or event data occur logically simultaneously and are routed to the different ports of the same `aperiodic`, `sporadic`, `timed`, or `hybrid` thread, the order of event handling is determined by the `Urgency` property associated with the ports.

(44) For a thread whose dispatch protocol is `background`, the thread is dispatched upon completion of its initialization entrypoint execution the first time it is active in a mode. The `Enabled` function is `true`. The `Wait_For_Dispatch` invariant is `t = 0`. The dispatch occurs immediately. If the `Dispatch_Trigger` property is set, then its execution is initiated through the arrival of an event or event data on one of those ports. In that case, the `Enabled` function is `true` for any `Dispatch` port `p ∈ Dispatch_Trigger : p ≠ ∅`.

(45) The different dispatch protocols can be summarized as follows:

Dispatch protocol	Dispatch condition
Periodic	Every period triggered by the dispatcher of the runtime system
Aperiodic	Triggered by the arrival of events, event data, subprogram calls
Sporadic	Triggered by the arrival of events, event data, subprogram calls with a minimum time difference of the specified period between dispatches
Timed	Triggered by the arrival of events, event data, subprogram calls with a timeout at the specified period.
Hybrid	Triggered by the arrival of events, event data, subprogram calls, as well as every period triggered by the dispatcher.

(46) Note that background threads do not have their current execution position reset on a mode switch. In other words, the background thread will resume execution from where it was previously suspended due to a mode switch.

(47) Note that background threads are suspended when not active in the current mode. If they have access to shared data components, they may have locked the resource at the time of suspension and potentially cause deadlock if active threads also share access to the same data component.

(48) A background thread is scheduled to execute such that all other threads' timing requirements are met. If more than one background thread is dispatched, the processor's scheduling protocol determines how such background threads are scheduled. For example, a FIFO protocol for background threads means that one background thread at a time is executed, while fair share means that all background threads will make progress in their execution.

(49) The `Overflow_Handling_Protocol` property for event or event data ports specifies the action to take when events arrive too frequently. These events are ignored, queued, or are treated as an error. The error treatment causes the currently active dispatch to be aborted, allowing it to clean up through the `Recover_Entrypoint` and then be redispatched. For more details on port queuing see section 8.3.3.

Examples

```
thread Prime_Reporter
features
  Received_Prime : in event data port Integer_Type;
properties
  Dispatch_Protocol => Timed;
end Prime_Reporter;

thread Prime_Reporter_One extends Prime_Reporter
features
  Received_Prime : refined to in event data port Integer_Type
    {Compute_Entrypoint => "Primes.On_Received_Prime_One";};
    -- function called on message-based dispatch
properties
  Period => 9 Sec; -- timeout period
  Priority => 45;
  Compute_Entrypoint => "Primes.Report_One";
  -- function called in case of timeout
end Prime_Reporter_One;
```

5.4.3 Thread Scheduling and Execution

(50) When a thread action is *performing thread computation* (see Figure 5), the execution of the thread's entrypoint source text code sequence is managed by a scheduler. This scheduler coordinates all thread executions on one processor as well as concurrent access to shared resources. While performing the execution of an entrypoint the thread can be *executing nominally* or *executing recovery* (see Figure 7). While executing an entrypoint a thread can be in one of five substates: ready, running, awaiting resource, awaiting return, and awaiting resume (see Figure 6).

(51) A thread initially enters the *ready* state. A scheduler selects one thread from the set of threads in the ready state to run on one processor according to a specified scheduling protocol. It ensures that only one thread is in the *running* state on a particular processor. If no thread is in the ready state, the processor is idle until a thread enters the ready state. A thread will remain in the running state until it completes execution of the dispatch, until a thread entering the ready state preempts it if the specified scheduling protocol prescribes preemption, until it blocks on a shared resource, or until an error occurs. In the case of completion, the thread transitions to the suspended *awaiting dispatch* state, ready to service another dispatch request. In the case of preemption, the thread returns to the ready state. In the case of resource blocking, it transitions to the *awaiting resource* state.

(52) Shared data is accessed in a critical region. Resource blocking can occur when a thread attempts enter a critical region while another thread is already in this critical region. In this case the thread enters the *Awaiting resource* state. A `Concurrency_Control_Protocol` property value associated with the shared data component determines the particular concurrency control mechanism to be used (see Section 5.1). The `Get_Resource` and `Release_Resource` service calls are provided to indicate the entry and exit of critical regions (see Section 5.1.1). When a thread completes execution it is assumed that all critical regions have been exited, i.e., access control to shared data has been released. Otherwise, the execution of the thread is considered erroneous.

(53) Subprogram calls to remote subprograms are synchronous or semi-synchronous. In the synchronous case, a thread in the running state enters the *awaiting return* state when performing a call to a subprogram whose service is performed by a subprogram in another thread. The service request for the execution of the subprogram is transferred to the remote subprogram request queue of a thread as specified by the `Actual_Subprogram_Call` property that specifies the binding of the subprogram call to a subprogram in another thread. When the thread executing the corresponding remote subprogram completes and the result is available to the caller, the thread with the calling subprogram transitions to the ready state. In the semi-synchronous case, the calling thread continues to execute concurrently until it awaits the result of the call (see service call `Await_Result` in Section 5.4.8).

(54) A background thread may be temporarily suspended by a mode switch in which the thread is not part of the new mode, as indicated by the **exit(Mode)** in Figure 6. In this case, the thread transitions to the *awaiting mode_entry* state. If the thread was in a critical region, it will be suspended once it releases all resources on exit of the critical region. A background thread resumes execution when it becomes part of the current mode again in a later mode switch. It then transitions from the *awaiting_mode_entry* state into the *ready* state.

(55) Execution of any of these entrypoints is characterized by actual execution time (c) and by elapsed time (t). Actual execution time is the time accumulating while a thread actually runs on a processor. Elapsed time is the time accumulating as real time since the arrival of the dispatch request. Accumulation of time for c and t is indicated by their first derivatives δc and δt. A derivative value of 1 indicates time accumulation and a value of 0 indicates no accumulation. Figure 6 shows the derivative values for each of the scheduling states. A thread accumulates actual execution time only while it is in the running state. The processor time, if any, required to switch a thread between the running state and any of the other states, which is specified in the `Thread_Swap_Execution_Time` property of the processor, is not accounted for in the `Compute_Execution_Time` property, but must be accounted for by an analysis tool.

(56) The execution time and elapsed time for each of the entrypoints are constrained by the entrypoint-specific `<entrypoint>_Execution_Time` and entrypoint-specific `<entrypoint>_Deadline` properties specified for the thread. If no entrypoint specific execution time or deadline is specified, those of the containing thread apply. There are three timing constraints:

- Actual execution time, c, will not exceed the maximum entrypoint-specific execution time.
- Upon execution completion the actual execution time, c, will have reached at least the minimum entrypoint-specific execution time.
- Elapsed time, t, will not exceed the entrypoint-specific deadline.

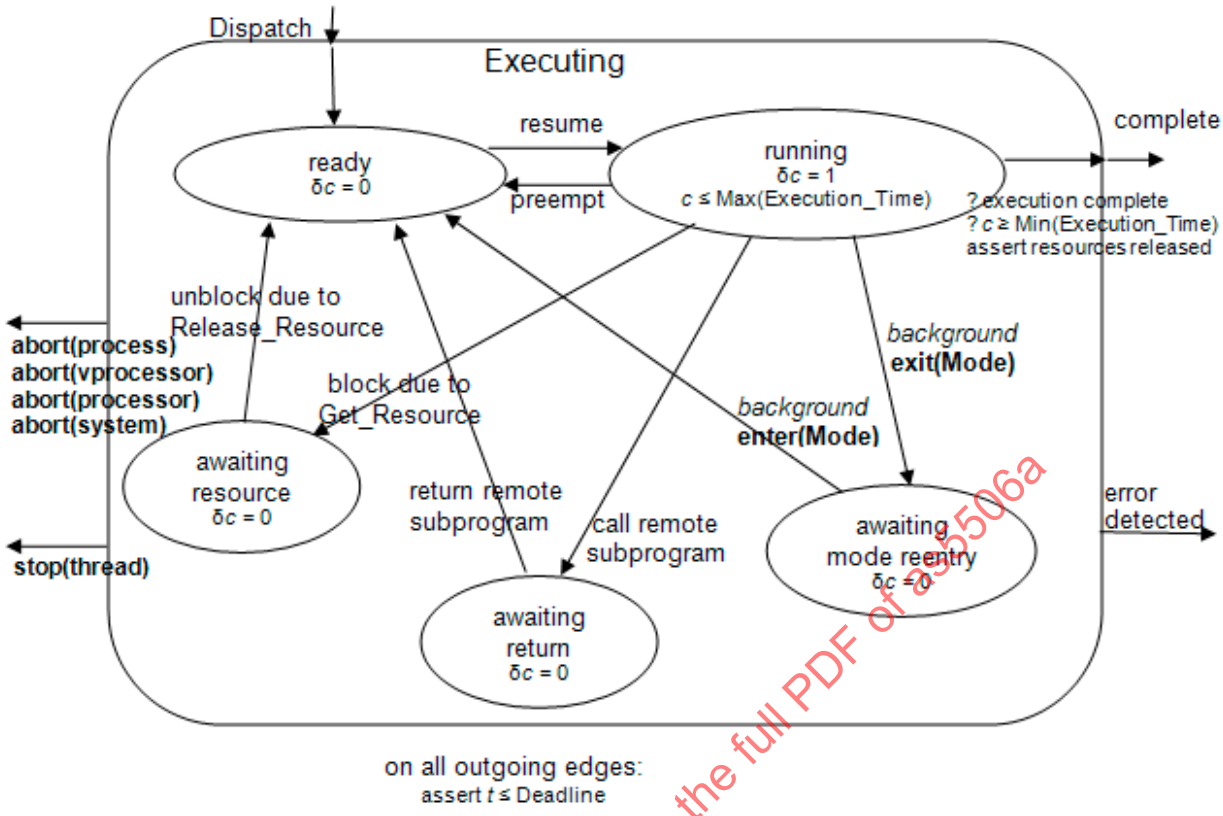


Figure 6 Thread Scheduling and Execution States

(57) Execution of a thread is considered anomalous when the timing constraints are violated. Each timing constraint may be enforced and reported as an error at the time, or it may be detected after the violation has occurred and reported at that time. The implementor of a runtime system must document how it handles timing constraints.

5.4.4 Execution Fault Handling

(58) A fault is defined to be an anomalous undesired change in thread execution behavior, possibly resulting from an anomalous undesired change in data being accessed by that thread or from violation of a compute time or deadline constraint. An error is a fault that is detected during the execution of a thread. Detectable errors are classified as *thread recoverable* errors, or *thread unrecoverable* errors.

(59) A *thread recoverable* error may be handled as part of normal execution by that thread, e.g., by exception handlers programmed in the source text of the application. The exception handler may propagate the error to an external handler by sending an event or event data through a port.

(60) If the thread recoverable error is not handled by the application, the thread affected by the error is given a chance to recover through the invocation of the thread's recover entrypoint. The recover entrypoint source text sequence has the opportunity to update the thread's application state. The recover entrypoint is assumed to have access to an error code through a runtime service call `Get_Error_Code`. The recover entrypoint may report the fact that it performed recovery through a user–defined port. Upon completion of the recover entrypoint execution, the performance of the thread's dispatch is considered complete. In the case of performing thread computation, this means that the thread transitions to the suspended await dispatch state (see Figure 5), ready to perform additional dispatches. Concurrency control on any shared resources must be released. If the recover entrypoint is unable to recover the error becomes a *thread unrecoverable* error. This thread-level fault handling in terms of thread scheduling states is illustrated in Figure 7.

(61) A thread recoverable error may occur during the execution of a remote subprogram call. In this case, the thread servicing the remote call is given a chance to recover as well as the thread that made the call.

(62) In the presence of a thread recoverable error, the maximum interval of time between the dispatch of a thread and

(63) its returning to the suspended awaiting dispatch state is the sum of the thread's compute deadline and its recover deadline. The maximum execution time consumed is the sum of the compute execution time and the recover execution time. In the case when an error is encountered during recovery, the same numbers apply.

(64) *Thread unrecoverable* errors are reported as event data through the `Error` port of the thread, where they can be communicated to a separate error handling thread for further analysis and recovery actions.

(65) A thread unrecoverable error causes the execution of a thread to be terminated prematurely without undergoing recovery. The thread unrecoverable error is reported as an error event through the predeclared `Error` event data port, if that port is connected. If this implicit error port is not connected, the error is not propagated and other parts of the system will have to recognize the fault through their own observations. In the case of a thread unrecoverable error, the maximum interval between the dispatch of the thread and its returning to the suspended awaiting dispatch state is the compute deadline, and the maximum execution time consumed is the compute execution time.

(66) For errors detected by the runtime system, error details are recorded in the data portion of the event as specified by the implementation. For errors detected by the source text, the application can choose its encoding of error detail and can raise an event in the source text. If the propagated error will be used to directly dispatch another thread or trigger a mode change, only an event needs to be raised. If the recovery action requires interpretation external to the raising thread, then an event with data must be raised. The receiving thread that is triggered by the event with data can interpret the error data portion and raise events that trigger the intended mode transition.

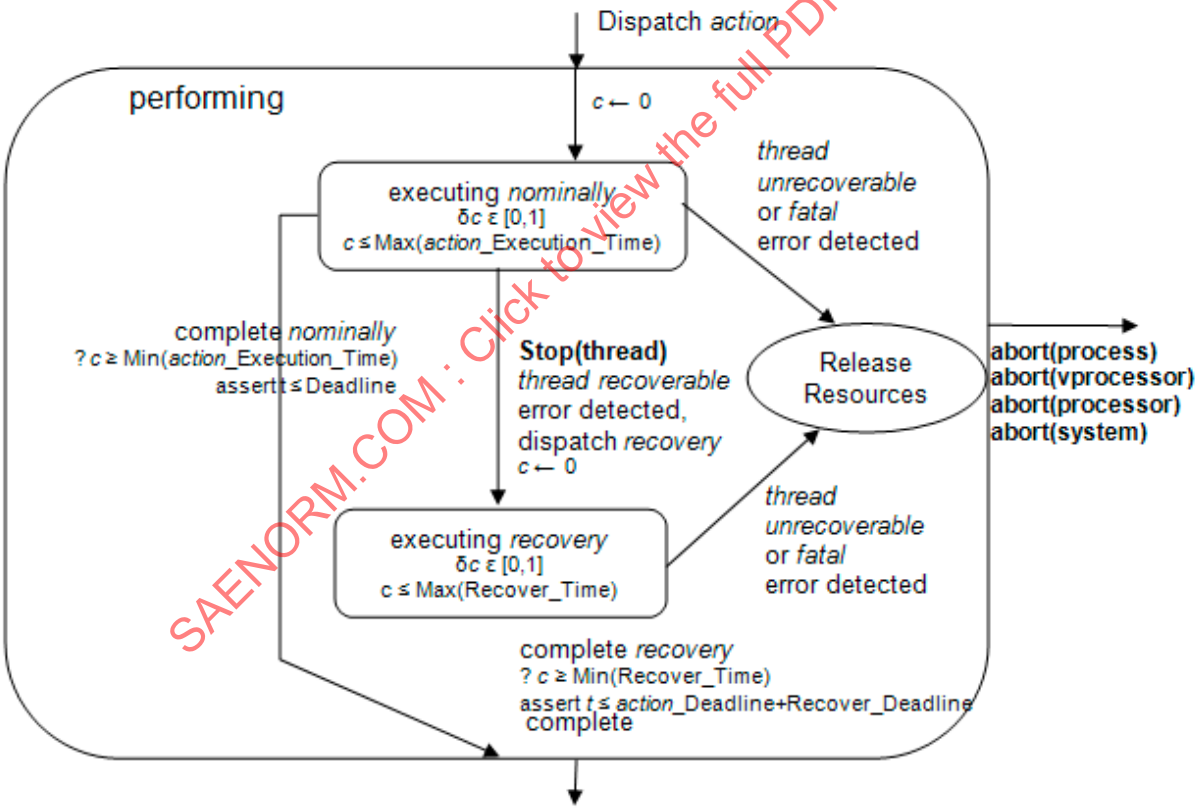


Figure 7 Performing Thread Execution with Recovery

(67) A timing fault during initialize, compute, activation, and deactivation entrypoint executions is considered to be a thread recoverable error. A timing fault during recover entrypoint execution is considered to be a thread unrecoverable error.

(68) If any error is encountered while a thread is executing a recover entrypoint, it is treated as a thread unrecoverable error. In other words, an error during recovery must not cause the thread to recursively re-enter the executing recovery state.

(69) If a fault is encountered by the application itself, it may explicitly raise an error through a `Raise_Error` service call on the `Error` port with the error class as parameter. This service call may be performed in the source text of any entrypoint. In the case of recover entrypoints, the error class must be *thread unrecoverable*.

(70) Faults may also occur in the execution platform. They may be detected by the execution platform components themselves and reported through an event or event data port, as defined by the execution platform component. They may go undetected until an application component such as a health monitoring thread detects missing health pulse events, or a periodic thread detects missing input. Once detected, such errors can be handled locally or reported as event data.

5.4.5 Thread Internal Modes and Mode Transitions

(71) A thread can have modes declared inside its type or implementation. They represent thread-internal execution paths and allow mode-specific property values to be associated with the thread. For example, the thread can have different execution times under different modes. Application source text (programming code) actually executes branch and merge points in various places of its code sequence and branches based on state values or based on input. In terms of the mode abstraction this means that the (mode) state at time of dispatch may affect the branch condition, the input to the thread execution may affect the branch condition, or a combination, or a change in the state value is the result of computation based on the previous value and/or input.

(72) A mode transition of a thread-internal mode may be implicit in that it is determined by the application source code of the thread. This source code may follow and execution sequence based on the content of thread input or by the value of a static data variable. In the former case, the current mode is determined at the time the thread input is determined, by default thread dispatch time. In the latter case, the value of the static data determines the current mode at the next dispatch. The effect is a possible change in mode-specific property values to reflect a change in source text internal execution behavior, e.g., a change in worst-case execution time, and in the entrypoint or call sequence to be executed at the next dispatch.

(73) Change of a thread-internal mode may be explicitly modeled by declaring a mode transition that names an incoming thread port, an event raised within a thread (declared as **self**.eventname), or names a subprogram call with an outgoing event. Change of a thread-internal mode may also be modeled through the Behavior Model Annex Document D. In this case, mode transitions are tracked by the runtime system to determine the most recent current mode for the next dispatch of a thread.

(74) If a higher fidelity behavioral model is desired, the Behavior Annex (Annex Document D), which uses the AADL modes as the initial set of states, can be used for more complex behavioral specifications. The final authority of the actual behavior of the source text is the program code itself.

5.4.6 System Synchronization Requirements

(75) An application system may consist of multiple threads. Each thread has its own hybrid automaton state with its own *c* and *t* variables. This results in a set of concurrent hybrid automata. In the concurrent hybrid automata model for the complete system, `ST` is a single real-valued variable shared by all threads that is never reset and whose rate is 1 in all states. `ST` is called the *reference timeline*.

(76) A set of periodic threads are said to be logically dispatched simultaneously at global real time *ST* if the order in which all exchanges of control and data at that dispatch event are identical to the order that would occur if those dispatches were exactly dispatched simultaneously in true and perfect real time. The *hyperperiod h* of a set of periodic threads is the next time *ST+h* at which they are logically dispatched simultaneously. The hyperperiod is the least common multiple of the periodic thread periods.

(77) An application system is said to be synchronized if the dispatch of all periodic threads contained in that application system occurs logically simultaneously at intervals of their hyperperiod. In a globally synchronous system `ST` is a global *reference time*, i.e., a single real-valued variable representing a global clock. It represents a single global *synchronization domain*.

(78) Within a synchronization domain, perfect synchronization may not occur in a actual system. There may always be clock error bounds in a distributed clock, and jitter in exactly when events (like a dispatch) would occur even with perfect clock interrupts due to things like non-preemptive blocking times (during which clock interrupts might be masked briefly). Within a synchronization domain, it is the responsibility of each physical implementation to take these imperfections into account when providing the synchronization domain for programmers (e.g., make sure the message transmission schedule includes enough margin for the message to arrive at the destination by the time it is needed, taking into account these various effects in the particular implementation).

5.4.7 Asynchronous Systems

(79) In a globally asynchronous system there are multiple *reference times*, i.e., multiple variables ST_j. They represent different *synchronization domains*. Any time related coordination and communication between threads, processors, and devices across different synchronization domains must take into account differences in the *reference time* of each of those synchronization domains.

(80) Reference times can be represented by instances of a predeclared device type called `Reference_Time`. Characteristics of this reference time, such as clock drift rate and maximum clock drift can be specified as properties of these instances. Processors, devices, buses, and memory can be assigned different reference times through the `Reference_Time` property. Similarly, application components can be assigned reference times to represent the fact that they may read the time, e.g., to timestamp data.

(81) The reference time for thread execution is determined by the reference time of the processor on which the thread executes. The reference time of communication between threads, devices, and processors is determined by the reference time of the source and destination, and the reference time of any execution platform component involved in the communication if it is time-driven.

(82) Message-passing semantics of communication and thread execution is represented by aperiodic threads whose dispatch is triggered by arrival of messages and message may be queued in the event data port. This communication paradigm is insensitive to time, thus, not affected by multiple synchronization domains.

(83) Data-stream semantics of communication and thread execution are represented by periodic threads and data ports. In this case the sampling of the input is sensitive to a common reference time between the source and the destination thread if the connections are immediate and delayed to ensure deterministic communication. Deterministic communication minimizes latency jitter, while non-deterministic communication can result in latency jitter in units of the sampling rate, the latter often leading to instability of latency sensitive applications such as control systems. In the case of sampling data port connections the non-deterministic nature of sampling accommodates different reference times. Similarly, a periodic thread may non-deterministically sample event ports and event data ports, e.g., a health monitor sampling an alarm queue.

(84) The `Allowed_Connection_Type` property of a bus specifies the types of connections supported by a bus. Buses that connect processors with different reference times may exclude immediate and delayed connections from their support if determinism cannot be guaranteed through a protocol.

(85) Mode switching requires time-sensitive coordination of deactivation and activation of threads and connections. There is the time ordering of events that request mode switching, and the coordination of switching modes in multiple modal subsystems as part of a single mode switch. Timed coordination can be guaranteed within one synchronization domain and may be feasible across synchronization domains with bounded time drift through appropriate protocols.

(86) Solutions have been devised to address this issue.

- ARINC653: Thread execution and communication within a partition is assumed to be within the same synchronization domain, Cross-partition communication is assumed to be message-based or (phase-)delayed for sampling ports. This assures that placement of partitions on different processors or at different parts of the timeline within one processor does not affect the timing. However, this delayed communication places a synchronicity requirement on those partitions that communicate with each other.

- Globally Asynchronous Locally Synchronous (GALS): This model reflects the fact that some systems cannot be globally synchronized, e.g., integrated modular avionics (IMA) system may consist of a collection of ARINC653 subsystems and interact via an ARINC664 network. In this case the burden is placed on the application system to deal with synchronicity within subsystems and asynchronicity across subsystems. This can be reflected in AADL by multiple synchronization domains and the requirement that data port connections across synchronization domains are sampled connections.
- Time Triggered Architecture (TTA): In this model a central communication medium provides a statically allocated time-division protocol and acts as a global reference time. Either part of the protocol provides reference time ticks to subsystems. Execution of subsystems can be aligned with the arrival of data at assigned time slots in the communication protocol to assure deterministic communication of data streams.
- Physically Asynchronous Logically Synchronous (PALS): In this model a logical protocol or application layer provides coordination of time-sensitive events across asynchronous subsystems. For example, the system may periodically re-synchronize clocks, thus, bound clock drift. This clock drift bound may be accommodated by appropriate time slack the same way jitter in a synchronous system is accommodated. Similarly, hand-shaking protocols may be used to coordinate less frequently occurring synchronization events, e.g., globally synchronous mode switching if required.

5.4.8 Runtime Support For Threads

(87) A standard set of runtime services are provided. The application program interface for these services is defined in the applicable source language annex of this standard. They are callable from within the source text. The following subprograms may be explicitly called by application source code, or they may be called by an AADL runtime system that is generated from an AADL model.

(88) The `Await_Dispatch` runtime service is called to suspend the thread execution at completion of its dispatch execution. It is the point at which the next dispatch resumes. The service call takes several parameters. It takes a `DispatchPort` list and an optional trigger condition function to identify the ports and the condition under which the dispatch is triggered. If the condition function is not present any of the ports in the list can trigger the dispatch. It takes a `DispatchedPort` as out parameter to return the port(s) that triggered the dispatch. It takes `OutputPorts` and `InputPorts` as port lists. `OutputPorts`, if present, identifies the set of ports whose sending is initiated at completion of execution, equivalent to an implicit `Send_Output` service call. `InputPorts`, if present, identifies the set of ports whose content is received at the next dispatch, equivalent to an implicit `Receive_Input` service call.

```
subprogram  Await_Dispatch
features
  -- List of ports whose output is sent at completion/deadline
  OutputPorts: in parameter <implementation-defined port list>;
  -- List of ports that can trigger a dispatch
  DispatchPorts: in parameter <implementation-defined port list>;
  -- list of ports that did trigger a dispatch
  DispatchedPort: out parameter < implementation-defined port list>;
  -- optional function as dispatch guard, takes port list as parameter
  DispatchConditionFunction: requires subprogram access;
  -- List of ports whose input is received at dispatch
  InputPorts: in parameter <implementation-defined port list>;
end Await_Dispatch;
```

(89) A `Raise_Error` runtime service shall be provided that allows a thread to explicitly raise a thread recoverable or thread unrecoverable error. `Raise_Error` takes an error type identifier as parameter.

```
subprogram  Raise_Error
features
```

```
    errorID: in parameter <implementation-defined error type>;
end Raise_Error;
```

(90) A `Get_Error_Code` runtime service shall be provided that allows a recover entrypoint to determine the type of error that caused the entrypoint to be invoked.

```
subprogram Get_Error_Code
features
    errorID: out parameter <implementation-defined error type>;
end Get_Error_Code;
```

(91) Subprograms have event ports but do not have an error port. If a `Raise_Error` is called, it is passed to the error port of the enclosing thread. If a `Raise_Error` is called by a remotely called subprogram, the error is passed to the error port of the thread executing the remotely called subprogram. The `Raise_Error` method is permitted to have an error identification as parameter value. This error identification can be passed through the error port as the data value, since the error port is defined as event data port.

(92) A `Await_Result` runtime service shall be provided that allows an application to wait for the result of a semi-synchronous subprogram call.

```
subprogram Await_Result
features
    CallID: in parameter <implementation-defined call ID>;
end Await_Result;
```

Processing Requirements and Permissions

(93) Multiple models of implementation are permitted for the dispatching of threads.

- One such model is that a runtime executive contains the logic reflected in Figure 5 and calls on the different entrypoints associated with a thread. This model naturally supports source text in a higher level domain language.
- An alternative model is that the code in the source text includes a code pattern that reflects the logic of Figure 5 through explicitly programmed calls to the standard `Await_Dispatch` runtime service, including a repeated call (while loop) to reflect repeated dispatch of the compute entrypoint code sequence.

(94) Multiple models of implementation are permitted for the implementation of thread entrypoints.

- One such model is that each entrypoint is a possibly separate function in the source text that is called by the runtime executive. In this case, the logic to determine the context of an error is included in the runtime system.
- A second model of implementation is that a single function in the source text is called for all entrypoints. This function then invokes an implementer-provided `Dispatch_Status` runtime service call to identify the context of the call and to branch to the appropriate code sequence. This alternative is typically used in conjunction with the source text implementation of the dispatch loop for the compute entrypoint execution.

(95) A method of implementing a system is permitted to choose how executing threads will be scheduled. A method of implementation is required to verify to the required level of assurance that the resulting schedule satisfies the period and deadline properties. That is, a method of implementing a system should schedule all threads so that the specified timing constraints are always satisfied.

(96) The use of the term “preempt” to name a scheduling state transition in Figure 6 does not imply that preemptive scheduling disciplines must be used; non-preemptive disciplines are permitted.

(97) Execution times associated with transitions between thread scheduling states, for example context swap times (specified as properties of the hosting processor), are not billed to the thread's actual execution time, i.e., are not reflected in the `Compute_Time` property value. However, these times must be included in a detailed schedulability model for a system. These times must either be apportioned to individual threads, or to anonymous threads that are introduced into the schedulability model to account for these overheads. A method of processing specifications is permitted to use larger compute time values than those specified for a thread in order to account for these overheads when constructing or analyzing a system.

(98) A method of implementing a system must support the periodic dispatch protocol. A method of implementation may support only a subset of the other standard dispatch protocols. A method of implementation may support additional dispatch protocols not defined in this standard.

(99) A method of implementing a system may perform loading and initialization activities prior to the start of system operation. For example, binary images of processes and initial data values may be loaded by permanently storing them in programmable memory prior to system operation.

(100) A method of implementing a system must specify the set of errors that may be detected at runtime. This set must be exhaustively and exclusively divided into those errors that are thread recoverable or thread unrecoverable, and those that are exceptions to be handled by language constructs defined in the applicable programming language standard. The set of errors classified as source language exceptions may be a subset of the exceptions defined in the applicable source language standard. That is, a method of implementation may dictate that a language-defined exceptional condition should not cause a runtime source language exception but instead immediately result in an error. For each error that is treated as a source language exception, if the source text associated with that thread fails to properly catch and handle that exception, a method of implementation must specify whether such unhandled exceptions are thread recoverable or thread unrecoverable errors.

(101) A consequence of the above permissions is that a method of implementing a system may classify all errors as thread unrecoverable, and may not provide an executing recovery scheduling state and transitions to and from it.

(102) A method of implementing a system may enforce, at runtime, a minimum time interval between dispatches of sporadic threads. A method of implementing a system may enforce, at runtime, the minimum and maximum specified execution times. A method of implementing a system may detect at runtime timing violations.

(103) A method of implementing a system may support handling of errors that are detected while a thread is in the suspended, ready, or blocked state. For example, a method of implementation may detect event arrivals for a sporadic thread that violate the specified period. Such errors are to be kept pending until the thread enters the executing state, at which instant the errors are raised for that thread and cause it to immediately enter the recover state.

(104) If alternative thread scheduling semantics are used, a thread unrecoverable error that occurs in the perform thread initialization state may result in a transition to the perform thread recovery state and thence to the suspended awaiting mode state, rather than to the thread halted state. The deadline for this sequence is the sum of the initialization deadline and the recovery deadline.

(105) If alternative thread scheduling semantics are used, a method of implementation may prematurely terminate threads when a system mode change occurs that does not contain them, instead of entering suspended awaiting mode. Any blocking resources acquired by the thread must be released.

(106) If alternative thread scheduling semantics are used, the load deadline and initialize deadline may be greater than the period for a thread. In this case, dispatches of periodic threads shall not occur at any dispatch time prior to the initialization deadline for that periodic thread.

(107) This standard does not specify which thread or threads perform virtual address space loading. This may be a thread in the runtime system or one of the application threads.

NOTES:

The deadline of a calling thread will impose an end-to-end deadline on all activities performed by or on behalf of that thread, including the time required to perform any remote subprogram calls made by that thread. The deadline property of a remotely called subprogram may be useful for scheduling methods that assign intermediate deadlines in the course of producing an overall end-to-end system schedule.

5.5 Thread Groups

(1) A thread group represents an organizational component to logically group threads contained in processes. The type of a thread group component specifies the features and required subcomponent access through which threads contained in a thread group interact with components outside the thread group. Thread group implementations represent the contained threads and their connectivity. Thread groups can have multiple modes, each representing a possibly different configuration of subcomponents, their connections, and mode-specific property associations. Thread groups can be hierarchically nested.

(2) A thread group does not represent a virtual address space nor does it represent a unit of execution. Therefore, a thread group must be directly or indirectly contained within a process.

Legality Rules

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

(L1) A thread group component type can contain provides and requires data access, as well as port, feature group, provides and requires subprogram access declarations, and provides and requires subprogram group access declarations. It can also contain flow specifications, modes subclauses, and property associations.

(L2) A thread group component implementation can contain abstract, data, subprogram, subprogram group, thread, and thread group subcomponent declarations.

(L3) A thread group implementation can contain a connections subclause, a flows subclause, a modes subclause, and properties subclause.

(L4) A thread group must not contain a subprogram calls subclause.

Standard Properties

```
-- Properties related to source text
Source_Text: inherit list of aadlstring
-- Inheritable thread properties
Synchronized_Component: inherit aadlboolean => true

Active_Thread_Handling_Protocol:
   inherit Supported_Active_Thread_Handling_Protocols => abort
```

```
Period: inherit Time
Deadline: inherit Time => Period
Dispatch_Offset: inherit Time
First_Dispatch_Time : inherit Time
-- Scheduling properties
Priority: inherit aadlinteger
Time_Slot: list of aadlinteger
Criticality: aadlinteger
-- execution time related properties
Reference_Processor: inherit classifier ( processor )
-- mode related properties
Resumption_Policy: enumeration ( restart, resume )
-- startup properties
Startup_Deadline: Time
Startup_Execution_Time: Time_Range
-- Properties specifying constraints for processor and memory binding
Allowed_Processor_Binding_Class:
   inherit list of classifier (processor, virtual processor, system)
Allowed_Processor_Binding: inherit list of reference (processor, virtual processor,
system)
Allowed_Memory_Binding_Class:
   inherit list of classifier (memory, system, processor)
Allowed_Memory_Binding: inherit list of reference (memory, system, processor)

Actual_Processor_Binding: inherit list of reference (processor, virtual processor)
Actual_Memory_Binding: inherit list of reference (memory)
Allowed_Connection_Binding_Class:
   inherit list of classifier(processor, virtual processor, bus, virtual bus, device,
memory)
Allowed_Connection_Binding: inherit list of reference (processor, virtual processor, bus,
virtual bus, device, memory)
Actual_Connection_Binding: inherit list of reference (processor, virtual processor, bus,
virtual bus, device, memory)
```

NOTES:

Property associations of thread groups are inheritable (see Section 11.3) by contained subcomponents. This means if a contained thread does not have a property value defined for a particular property, then the corresponding property value for the thread group is used.

Semantics

(3) A thread group allows threads contained in processes to be logically organized into a hierarchy. A thread group type declares the features and required subcomponent access through which threads contained in a thread group can interact with components declared outside the thread group.

(4) Thread groups may contain subprogram subcomponents and subprogram groups The code of such subprograms and subprogram groups resides in the address space of the containing process. The subprograms may be called by threads contained in the thread group. The subprograms may also be called from outside the thread group if made accessible through a provides subprogram access declaration or subprogram group access declaration.

(5) Thread groups may contain data components. They represent state that may be shared between threads inside the thread group through access connections to the requires data access features of those threads, and shared outside the thread group through provides data access features of the thread group.

(6) A thread group implementation contains threads and thread groups. Thread group nesting permits threads to be organized hierarchically. A thread group implementation also contains connections to specify the interactions between the contained subcomponents and modes to represent different configurations of subsets of subcomponents and connections as well as mode-specific property associations.

5.6 Processes

(1) A process represents a virtual address space, i.e., it represents a space partition unit whose boundaries are enforced at runtime. The `Runtime_Protection` process property indicates whether runtime protection is disabled. The virtual address space contains the program formed by the source text associated with the process and its subcomponents. Threads of a process must be explicitly declared.

Legality Rules

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

(L1) A process component type can contain port, feature group, provides and requires data access, provides and requires subprogram access declarations, and provides and requires subprogram group access declarations. It can also contain flow specifications, modes subclause, and property associations.

(L2) A process component implementation can contain abstract, data, subprogram, subprogram group, thread, and thread group subcomponent declarations.

(L3) A process implementation can contain a connections subclause, a flows subclause, a modes subclause, and a properties subclause.

Consistency Rules

(C1) The complete source text associated with a process component must form a complete and legal program as defined in the applicable source language standard. This source text shall include the source text that corresponds to the complete set of subcomponents in the process's containment hierarchy along with the data and subprograms that are referenced by required subcomponent declarations.

Standard Properties

```
-- Runtime enforcement of virtual address space boundary
Runtime_Protection : inherit aadlboolean
-- Properties related to source text
Source_Text: inherit list of aadlstring
Source_Language: inherit list of Supported_Source_Languages
-- Properties related to virtual address space loading
Load_Time: Time_Range
Load_Deadline: Time
-- Inheritable thread properties
Synchronized_Component: inherit aadlboolean => true
Active_Thread_Handling_Protocol:
   inherit Supported_Active_Thread_Handling_Protocols => abort
Period: inherit Time
Deadline: inherit Time => Period
Dispatch_Offset: inherit Time
-- execution time related properties
Reference_Processor: inherit classifier ( processor )
-- Scheduling related properties
Priority: inherit aadlinteger
-- mode related properties
Resumption_Policy: enumeration ( restart, resume )
Deactivation_Policy: enumeration (inactive, unload) => inactive
-- process initialization
Startup_Deadline: Time
Startup_Execution_Time: Time_Range
-- Properties specifying constraints memory binding
Allowed_Processor_Binding_Class:
   inherit list of classifier (processor, virtual processor, system)
Allowed_Processor_Binding: inherit list of reference (processor, virtual processor,
system)
Actual_Processor_Binding: inherit list of reference (processor, virtual processor)
Allowed_Connection_Binding_Class:
   inherit list of classifier(processor, virtual processor, bus, virtual bus, device,
memory)
Allowed_Connection_Binding: inherit list of reference (processor, virtual processor, bus,
virtual bus, device, memory)
Actual_Connection_Binding: inherit list of reference (processor, virtual processor, bus,
virtual bus, device, memory)
Allowed_Memory_Binding_Class:
   inherit list of classifier (memory, system, processor)
```

```
Allowed_Memory_Binding: inherit list of reference (memory, system, processor)
Actual_Memory_Binding: inherit list of reference (memory)
```

Semantics

(2) Every process has its own virtual address space. This address space provides access to source code and data associated with the process and all its contained components. This address space boundary is by default enforced at runtime, but can be disabled through the `Runtime_Protection` property.

(3) Threads contained in a process execute within the virtual address space of the process.

(4) Processes may contain subprogram subcomponents. The code of such subprograms resides in the address space of the process. The calling semantics to such subprograms are defined in Section 5.2.

(5) A process may contain mode declarations. In this case, each mode can represent a different configuration of contained threads, their connections, and mode-specific property associations. The transition between modes is determined by the mode transition declarations and is triggered by the arrival of *mode transition trigger events* (see Sections 12 and 13.6).

(6) The associated source text for each process is compiled and linked to form binary images in accordance with the applicable source language standard. These binary images must be loaded into memory before any thread contained in a process can execute, i.e., enter its *perform thread initialization* state.

(7) The time to load binary images into the virtual address space of a process is bounded by the `Load_Deadline` and `Load_Time` properties. The failure to meet these timing requirements is considered an error.

(8) The process states, transitions, and actions are illustrated in Figure 8. Once a processor of an execution platform is started, binary images making up the virtual address space of processes bound to the processor are ready to be loaded, which is indicated by **started(processor)**. If the process is bound to a virtual processor of a processor, then process loading begins when the virtual processor is started, which is indicated by **started(vprocessor)**. Loading may take zero time for binary images that have been preloaded in ROM, PROM, or EPROM. Completion of loading, which is indicated by **loaded(process)**, triggers threads to be initialized (see Figure 5).

(9) A process, i.e., its contained threads, can be stopped (also known as a deferred abort), which is indicated by **stop(process)** , or by stopping the virtual processor or processor to which the process is bound. A process is considered stopped when all threads of the process are halted, are awaiting a dispatch, or are not part of the current mode and have executed their finalize entrypoint.

(10) A process, i.e., its contained threads, can be aborted, which is indicated by **abort(process)**. In this case, all contained threads terminate their execution immediately and release any resources (see Figure 5, Figure 6, and Figure 7).

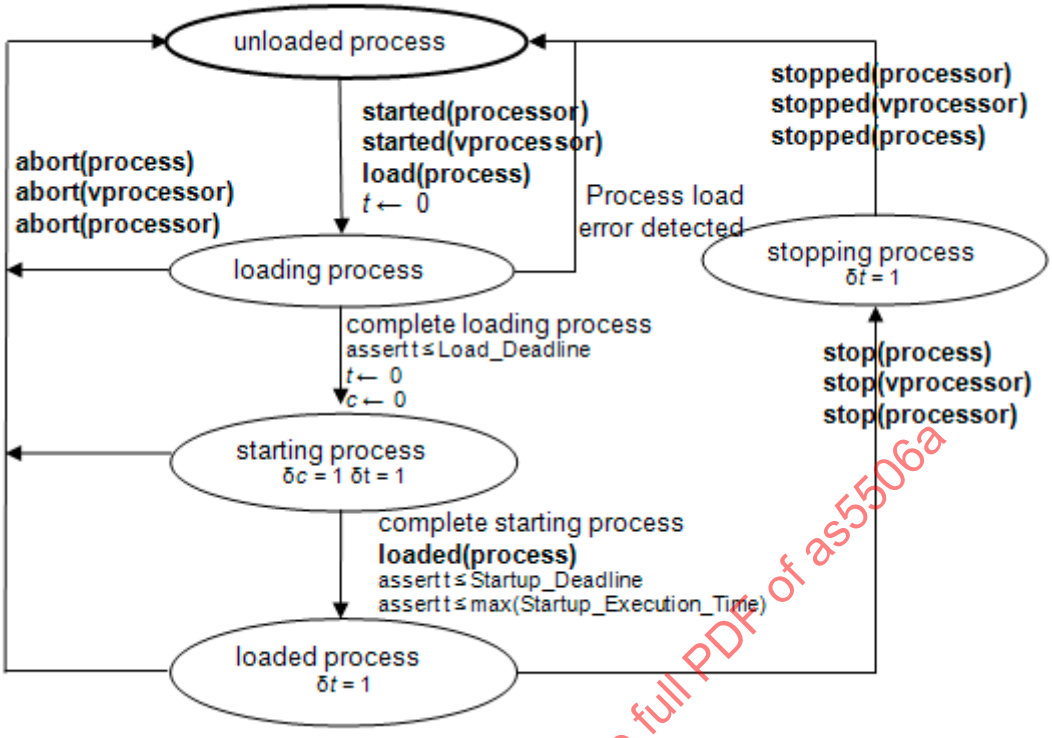


Figure 8 Process States and Actions

Processing Requirements and Permissions

(11) A method of implementation must link all associated source text for the complete set of subcomponents of a process, including the process component itself and all actual subcomponents specified for required subcomponents. This set of source compilation units must form a single complete and legal program as defined by the applicable source language standard. Linking of this set of source compilation units is performed in accordance with the applicable source language standard for the process.

(12) If the applicable source language standard used to implement a component permits a mixture of source languages, then subcomponents may have different source language property values.

(13) This standard permits dynamic virtual memory management or dynamic library linking after process loading has completed and thread execution has started. However, a method for implementing a system must assure that all deadline properties will be satisfied to the required level of assurance for each thread.

NOTES:

An AADL process represents only a virtual address space and requires at least one explicitly declared thread subcomponent in order to be executable. The explicitly declared thread in AADL allows for unambiguous specification of port connections to threads. In contrast, a POSIX process represents both a protected address space and an implicit thread.

6 Execution Platform Components

(1) This section describes the categories of execution platform components that represent computing hardware: processor, virtual processor, memory, bus, virtual bus; and the physical environment: device.

(2) Processors can execute threads. Processors can contain memory subcomponents. Processors can access memories and communicate with devices and other processors over buses. Threads, thread groups, and processes are bound to processors.

(3) Virtual processors are logical execution platform elements that can execute threads. Threads, thread groups, and processes can be bound to virtual processors. Virtual processors must be bound to or contained in processors. This determines the binding of threads to processors.

(4) Memories represent randomly addressable storage capable of storing binary images in the form of data and code. Memories can be accessed by executing threads.

(5) Buses support physical communication between processors, devices, and memories. A bus provides the resources necessary to perform exchanges of control and data as specified by connections. These resources include bandwidth and protocols to perform the exchange. A connection may be bound to a sequence of buses and intermediate processors and devices in a manner that is analogous to the binding of threads to processors.

(6) Virtual buses represent a logical bus abstraction to model protocols and virtual channels. Virtual buses can be contained in or bound to processors and buses. Connections can specify that they require specific protocols, or certain quality of service from protocols of platform components they are bound to.

(7) Devices represent entities of the physical environment, e.g., an engine, or entities that interface with the physical environment, e.g., a sensor or an actuator. A device can interact with application software components through their port and provides subprogram access features. A device may interact with other execution platform components through bus access connections. A device may achieve its functionality through device internal software or may require device driver software to be executed by a processor. Binary images or threads cannot be bound to devices.

(8) Processors may include software that implements the capability of the processor to schedule and execute threads and other services of the processor. Its source text and data in the form of binary images will be bound to memories accessible from that processor. The resource requirements of this software are reflected in processor properties.

(9) Execution platform components can be assembled into execution platform systems, i.e., into systems of execution platform components to model complex physical computing hardware components and software/hardware computing systems, through the use of system components (see Section 7.1). The execution platform systems and their components may denote physical computing hardware for example, memory to represent a hard disk or RAM. Execution platform systems may also model abstracted storage, for example, a device or memory to represent a database, depending on the purpose of the model.

(10) The hardware represented by the execution platform components may be modeled in a hardware description or simulation language. Alternatively, it may be represented using configuration data for programmable logic devices. A simulation may be used to characterize the components. Such descriptions can be associated with the component by property association.

(11) Execution platform components can represent high-level abstractions of physical and computing components. A detailed AADL model of their implementation can be represented by system implementations that are associated with the execution platform component by property (see Section 14). This effectively models a layered system architecture.

6.1 Processors

(1) A processor is an abstraction of hardware and software that is responsible for scheduling and executing threads and virtual processors that are bound to it. A processor also may execute driver software that is declared as part of devices that can be accessed from that processor. Processors may contain memories and may access memories and devices via buses.

Legality Rules

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

(L1) A processor component type can contain port, feature group, provides subprogram access, provides subprogram group access, and bus access declarations. It may contain flow specifications, a modes subclause, as well as property associations.

(L2) A processor component implementation can contain declarations of memory, bus, virtual bus, virtual processor, and abstract subcomponents.

(L3) A processor implementation can contain a modes subclause, flows subclause, and a properties subclause.

(L4) A processor implementation can contain bus access connection declarations.

(L5) A processor implementation must not contain a subprogram calls subclause.

Standard Properties

```
-- Hardware description properties
Hardware_Description_Source_Text: inherit list of aadlstring
Hardware_Source_Language: Supported_Hardware_Source_Languages
-- Properties related to source text that provides thread scheduling services
Source_Text: inherit list of aadlstring
Source_Language: inherit list of Supported_Source_Languages
Source_Code_Size: Size
Source_Data_Size: Size
Source_Stack_Size: Size
Allowed_Memory_Binding_Class:
   inherit list of classifier (memory, system, processor)
Allowed_Memory_Binding: inherit list of reference (memory, system, processor)
Allowed_Memory_Binding: inherit list of reference (memory, system, processor)
-- Processor initialization properties
```

```
Startup_Deadline: Time
Startup_Execution_Time: Time_Range
-- Properties specifying provided thread execution support
Thread_Limit: aadlinteger 0 .. Max_Thread_Limit
Allowed_Dispatch_Protocol: list of Supported_Dispatch_Protocols
Allowed_Period: list of Time_Range
Scheduling_Protocol: inherit list of Supported_Scheduling_Protocols
Scheduler_Quantum : inherit Time
Slot_Time: Time
Frame_Period: Time
-- acceptable priority value on threads and mapping into RTOS priority values
Priority_Range: range of aadlinteger
Priority_Map: list of Priority_Mapping
Process_Swap_Execution_Time: Time_Range
Thread_Swap_Execution_Time: Time_Range
Supported_Source_Language: list of Supported_Source_Languages
-- Scaling of processor speed
Scaling_Factor : inherit aadlreal
Reference_Processor: inherit classifier ( processor )
-- Properties related to data movement in memory
Assign_Time: record (
    Fixed: Time_Range;
    PerByte: Time_Range; )
  -- Properties related to the hardware clock
Clock_Jitter: Time
Clock_Period: Time
Clock_Period_Range: Time_Range
-- Protocol support
Provided_Virtual_Bus_Class : inherit list of classifier (virtual bus)
Provided_Connection_Quality_Of_Service : inherit list of Supported_Connection_QoS
-- mode related properties
Resumption_Policy: enumeration ( restart, resume )
Deactivation_Policy: enumeration (inactive, unload) => inactive
-- Virtual machine layering
Implemented_As: classifier ( system implementation )
```

NOTES:

The above is the list of the predefined processor properties. Additional processor properties may be declared in user-defined property sets. Candidates include properties that describe capabilities and accuracy of a synchronized clock, e.g., drift rates, differences across processors.

Semantics

(2) A processor is the execution platform component that is capable of scheduling and executing threads. Threads will be bound to a processor for their execution that supports the dispatch protocol required by the thread. The `Allowed_Dispatch_Protocol` property specifies the dispatch protocols that a processor supplies.

(3) Support for thread execution may be embedded in the processor hardware or it may require software that implements processor functionality such as thread scheduling, e.g., an operating system kernel or other software virtual machine. Such software must be bound to a memory component that is accessible to the processor via the `Actual_Memory_Binding` property. Services provided by such software can be called through the provides subprogram access features of a processor.

(4) The code that threads execute and the data they access must be bound to a memory component that is accessible to the processor on which the thread executes.

(5) If a processor executes device driver software associated with a device, then the processor must have access to the corresponding device component.

(6) Flow specifications model logical flow through processors. For example, they may represent requests for operating system services through subprograms or ports.

(7) The hardware description source text property may provide a reference to source text that is a model of the hardware in a hardware description language. This provides support for the simulation of hardware.

(8) Modes allow processor components to have different property values under different operational processor modes. Modes may be used to specify different runtime selectable configurations of processor implementations.

(9) Processor states and transitions are illustrated in the hybrid automaton shown in Figure 9. The labels in this hybrid automaton interact with labels in the system hybrid automaton (see Figure 22), the virtual processor hybrid automaton (see Figure 10), and the process hybrid automaton (see Figure 8).

- The initial state of a processor is *stopped*.
- When a processor is started, it enters the *processor starting* state. In this state, the processor hardware is initialized and any processor software that provides thread scheduling functionality is loaded into memory and initialized. Note that the virtual address space load images of processes may already have been loaded as part of a single load image that includes the processor or virtual processor software.
- Once the processor and its software for handling virtual processors or processes is initialized, the processor transitions to the *processor operational* state with **started(processor)**. At this point virtual processor, process, and in turn thread initialization will start.

(10) While operational, a processor may be in different modes with different processing characteristics reflected in appropriate property values.

(11) As a result of a processor abort, any threads bound to the processor are aborted, as indicated by **abort(processor)** in the hybrid automaton in Figure 9 and in the hybrid automata figures in Sections 5.4 and 5.6. After that any virtual processor bound to or contained in a processor is aborted.

(12) A stop processor request results in a transition to the *processor stopping* state at the next hyperperiod of the periodic threads bound to the processor or to its virtual processors. The length of the hyperperiod can be reduced by using the `Synchronized_Components` property to minimize the number of periodic threads that must be synchronized within the hyperperiod (see Section 12).

(13) When the next hyperperiod begins, the virtual processors and processes with threads bound to the processor are informed about the stoppage request, as indicated by **stop(processor)** in the hybrid automaton in Figure 9. The process hybrid automaton (see Figure 8) in turn causes the thread hybrid automaton to respond, as indicated with **stop(processor)** in the hybrid automaton in Figure 5. In this case, any threads bound to the processor are permitted to complete their dispatch executions and perform any finalization before the process is stopped, which is indicated by **stopped(process)** in Figure 8.

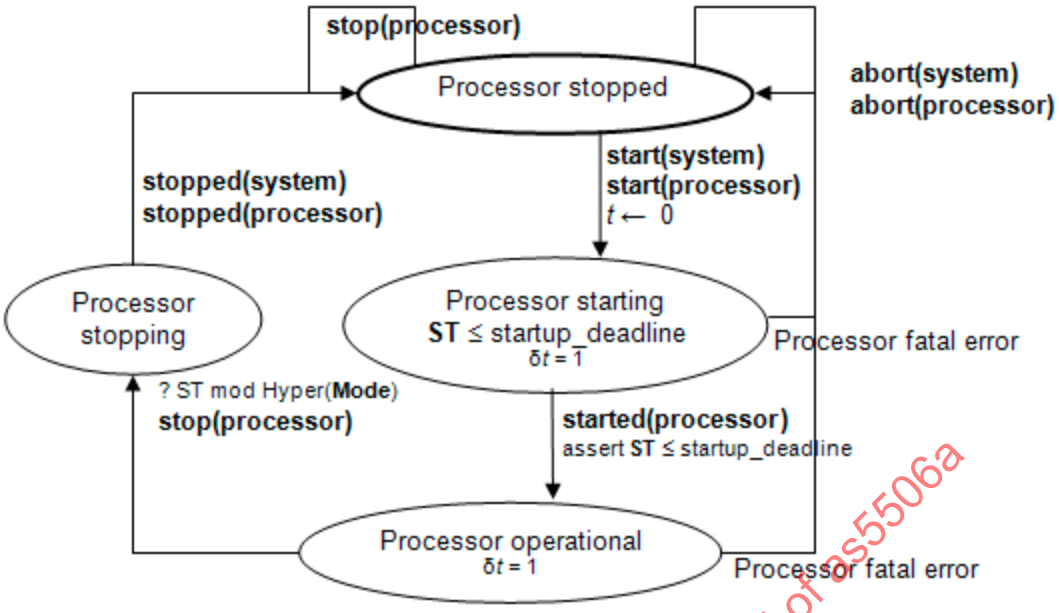


Figure 9 Processor States and Actions

(14) A processor may have ports through which it reports information to the application software, e.g., to report error conditions. Those ports are identified in port connection declarations by the reserved word **processor** (see Section 9.1).

(15) A processor may have provide subprogram or subprogram group access to represent services that can be called by the application. A subprogram call identifies such a service by the subprogram classifier and the binding to the specific processor is implicit through the actual processor binding of the thread that contains the call.

(16) A processor component can include protocols in its abstraction. These protocols can be explicitly modeled as virtual bus subcomponents to satisfy protocol requirements by connections. The fact that a protocol is supported by a processor is specified by a `Provided_Virtual_Bus_Class` property.

(17) A processor can contain a bus subcomponent that it makes externally accessible. This models a bus that is managed by the processor and other components can connect to it. In this case the processor is implicitly connected to this bus.

(18) Different processors may be different execution speeds. This affects the execution time specified for threads and subprograms. The **in binding** statement of property associations permits processor type specific execution times to be declared. The execution time of a thread or subprogram can also be scaled accordsing to scaling factors between different processors. The `Scaling_Factor property` specifies the scaling factor with respect to a specified `Reference_Processor`.

(19) The processor is an abstraction of a hardware processor and possibly a runtime system. If it is desirable, the internals of the processor can be modeled by AADL as a system in its own right, i.e., an application system and an execution platform. This system implementation description can be associated with the device component declaration by the `Implemented_As` property (see Section 14).

Processing Requirements and Permissions

(20) A method of implementation is not required to monitor the startup deadline and report an overflow as an error.

(21) A method of implementation may choose to enforce thread deadlines and maximum compute execution time. Violations are reported as thread recoverable errors.

6.2 Virtual Processors

(1) A virtual processor represents a logical resource that is capable of scheduling and executing threads and other virtual processors bound to them. Virtual processors can be declared as a subcomponent of a processor or another virtual processor, i.e., they are implicitly bound to the processor or virtual processor they are contained in. Virtual processors can also be declared separately, that is as a subcomponent of a system component, and explicitly bound to a processor or virtual processor. This allows virtual processors to represent hierarchical schedulers that schedule thread and/or virtual processors bound to them. In a fully bound system every virtual processor and thread is eventually bound to a physical processor.

Legality Rules

Category	Type	Implementation
virtual processor	Features: • provides subprogram access • provides subprogram group access • port • feature group • feature Flow specifications: yes Modes: yes Properties: yes	Subcomponents: • virtual processor • virtual bus • abstract Subprogram calls: no Connections: no Flows: yes Modes: yes Properties: yes

(L1) A virtual processor component type can contain port, feature group, provides subprogram access, and subprogram group access declarations. It may contain flow specifications, a modes subclause, as well as property associations.

(L2) A virtual processor component implementation can contain declarations of virtual bus, virtual processor, and abstract subcomponents.

(L3) A virtual processor implementation can contain a modes subclause, flows subclause, and a properties subclause.

(L4) A virtual processor implementation must not contain a subprogram calls subclause.

Consistency Rules

(C1) In a fully bound system every virtual processor must be directly or indirectly bound to, or directly or indirectly contained in a physical processor.

(C2) In a fully deployed system a requires virtual bus binding of a virtual processor specified by the `Required_Virtual_Bus_Class` property must be satisfied by binding the virtual processor to a virtual processor or processor that provides this virtual bus. It is also satisfied if the virtual processor is contained in a processor and the respective virtual bus is bound to the processor.

Standard Properties

```
-- Properties related to source text that provides thread scheduling services
Source_Text: inherit list of aadlstring
Source_Language: inherit list of Supported_Source_Languages
Source_Code_Size: Size
Source_Data_Size: Size
Source_Stack_Size: Size
Allowed_Memory_Binding_Class:
   inherit list of classifier (memory, system, processor)
Allowed_Memory_Binding: inherit list of reference (memory, system, processor)
```

```
Actual_Memory_Binding: inherit list of reference (memory)
Allowed_Processor_Binding_Class:
   inherit list of classifier (processor, virtual processor, system)
Allowed_Processor_Binding: inherit list of reference (processor, virtual processor,
system)
Actual_Processor_Binding: inherit list of reference (processor, virtual processor)
-- Virtual processor initialization properties
Startup_Execution_Time: Time_Range
Startup_Deadline: Time
-- Properties specifying provided thread execution support
Thread_Limit: aadlinteger 0 .. Max_Thread_Limit
Allowed_Dispatch_Protocol: list of Supported_Dispatch_Protocols
Allowed_Period: list of Time_Range
Scheduling_Protocol: inherit list of Supported_Scheduling_Protocols
Slot_Time: Time
Frame_Period: Time
-- acceptable priority value on threads and mapping into RTOS priority values
Priority_Range: range of aadlinteger
Priority_Map: list of Priority_Mapping
Process_Swap_Execution_Time: Time_Range
Thread_Swap_Execution_Time: Time_Range
Supported_Source_Language: list of Supported_Source_Languages
-- Properties of the dispatch characteristics of this virtual processor
Period: inherit Time
Dispatch_Protocol: Supported_Dispatch_Protocols
Execution_Time: Time
-- Protocol support
Provided_Virtual_Bus_Class : inherit list of classifier (virtual bus)
Provided_Connection_Quality_Of_Service : inherit list of Supported_Connection_QoS
-- mode related properties
Resumption_Policy: enumeration ( restart, resume )
Deactivation_Policy: enumeration (inactive, unload) => inactive
-- Virtual machine layering
Implemented_As: classifier ( system implementation )
```

NOTES:

The above is the list of the predefined virtual processor properties. Additional processor properties may be declared in user-defined property sets. Candidates include properties that describe capabilities and accuracy of a synchronized clock, e.g., drift rates, differences across processors.

Semantics

(2) A virtual processor is the logical execution platform component that is capable of scheduling and executing threads and other virtual processors. Threads and virtual processors will be bound to a virtual processor or processor for their execution. The `Allowed_Dispatch_Protocol` property specifies the dispatch protocols that a virtual processor supplies, i.e., only threads or virtual processors whose dispatch protocol is supported can be bound.

(3) Support for thread execution may require software that implements virtual processor functionality such as thread scheduling, e.g., an operating system kernel or other software virtual machine. Such software must be bound to a memory component that is accessible to the processor via the `Actual_Memory_Binding` property. Services provided by such software can be called through the provides subprogram access features of a virtual processor.

(4) A virtual processor component can include protocols in its abstraction. These protocols can be explicitly modeled as virtual bus subcomponents to satisfy protocol requirements by connections. The fact that a protocol is supported by a virtual processor can also be specified by a `Provided_Virtual_Bus_Class` property.

(5) A virtual processor can be declared as a subcomponent of a processor or another virtual processor; it can also be declared separately in a system component and then bound to a processor or another virtual processor. The difference between the two uses of virtual processor is that in the case of the subcomponent of a processor or virtual processor the binding of the virtual processor is implicit in the containment relationship.

(6) Flow specifications model logical flow through virtual processors. For example, they may represent requests for operating system services through subprograms or ports.

(7) Modes allow virtual processor components to have different property values under different operational virtual processor modes. Modes may be used to specify different runtime selectable configurations of virtual processor implementations.

(8) Virtual processor states and transitions are illustrated in the hybrid automaton shown in Figure 10. The hybrid automaton of a virtual processor interacts with the hybrid automaton of the processor or virtual processor that it is bound to. A virtual processor is permitted to initialize itself after the processor and any virtual processors are initialized, and before any processes or threads are initialized. Similarly, virtual processors are stopped after threads, processes, and virtual processors contained in them are stopped.

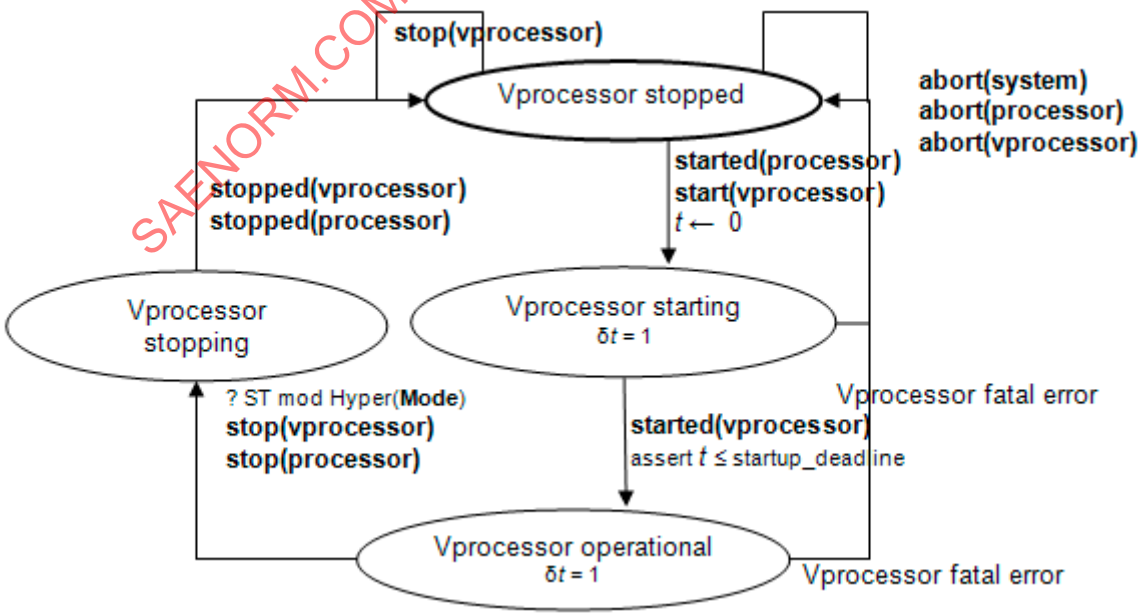


Figure 10 Virtual Processor States and Actions

(9) The virtual processor is an logical abstraction of a processor. If it is desirable, the internals of the virtual processor can be modeled by AADL as an application system in its own right. For example, a virtual processor may represent an operating system that can be described in terms of processes and threads. This system implementation description can be associated with the device component declaration by the `Implemented_As` property (see Section 14).

6.3 Memory

(1) A memory component represents an execution platform component that stores code and data binaries. This execution platform component consists of hardware such as randomly accessible physical storage, e.g., RAM, ROM, or more complex permanent storage such as disks, reflective memory, or logical storage. Memories have properties such as the number and size of addressable storage locations. Subprograms, data, and processes – reflected in binary images - are bound to memory components for access by processors when executing threads. A memory component may be contained in a processor or may be accessible from a processor via a bus.

Legality Rules

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

(L1) A memory type can contain bus access declarations, feature groups, a modes subclause, and property associations. It must not contain flow specifications.

(L2) A memory implementation can contain abstract, memory, and bus subcomponent declarations.

(L3) A memory implementation can contain a modes subclause and property associations.

(L4) A memory implementation can contain bus access connection declarations if it contains a memory subcomponent that requires bus access.

(L5) A memory implementation must not contain flows subclause, or subprogram calls subclause.

Standard Properties

```
-- Properties related memory as a resource and its access
Memory_Protocol: enumeration (read_only, write_only, read_write) => read_write
Word_Size: Size => 8 bits
Byte_Count: aadlinteger 0 .. Max_Word_Count
Word_Space: aadlinteger 1 .. Max_Word_Space => 1
Base_Address: aadlinteger 0 .. Max_Base_Address
Read_Time: record (
    Fixed: Time_Range;
    PerByte: Time_Range; )
Write_Time: record (
    Fixed: Time_Range;
```

```
    PerByte: Time_Range; )
-- Hardware description properties
Hardware_Description_Source_Text: inherit list of aadlstring
Hardware_Source_Language: Supported_Hardware_Source_Languages
-- mode related properties
Resumption_Policy: enumeration ( restart, resume )
-- Virtual machine layering
Implemented_As: classifier ( system implementation )
```

Semantics

(2) Memory components are used to store binary images of source text, i.e., code and data. These images are loaded into memory representing the virtual address space of a process and are accessible to threads contained in the respective processes bound to the processor. Such access is possible if the memory is contained in this processor or is accessible to this processor via a shared bus component. Loading of binary images into memory may occur during processor startup or the binary images may have been preloaded into memory before system startup. An example of the latter case is PROM or EPROM containing binary images.

(3) A memory is accessible from a processor if the memory is contained in a processor or is connected via a shared bus component and the `Allowed_Physical_Access` property value for that bus includes `Memory_Access`.

(4) Memory components can have different property values under different operational modes.

(5) The memory component is intended to be an abstraction of a physical storage component. This can be a memory component such as RAM, or it can represent a more abstract storage component such as a map database. If it is desirable, the internals of the memory can be modeled by AADL as a system in its own right, i.e., an application system and an execution platform. For example, a map data base as a memory component can be modeled as a set of processors and disks as well as the database software. This system implementation description can be associated with the memory component declaration by the `Implemented_As` property (see Section 14).

6.4 Buses

(1) A bus component represents an execution platform component that can exchange control and data between memories, processors, and devices. This execution platform component represents a communication channel, typically hardware together with communication protocols. Supported communication protocols can be explicitly modeled through virtual buses (see Section 6.5).

(2) Processors, devices, and memories can communicate by accessing a shared bus. Such a shared bus can be located in the same system implementation as the execution platform components sharing it or higher in the system hierarchy. Memory, processor, and device types, as well as the system type of systems they are contained in, can declare a need for access to a bus through a requires bus reference.

(3) Buses can be connected directly to other buses by one bus requiring access to another bus. Buses connected in such a way can have different bus classifier references.

(4) Connections between software components that are bound to different processors transmit their information across buses whose protocol supports the respective connection category.

Legality Rules

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

(L1) A bus type can have requires bus access declarations, a modes subclause, and property associations.

(L2) A bus type must not contain any flow specifications.

(L3) A bus implementation can contain virtual bus and abstract subcomponent declarations.

(L4) A bus implementation can contain a modes subclause and property associations.

(L5) A bus implementation must not contain a connections subclause, flows subclause, or subprogram calls subclause.

Standard Properties

```
-- Properties specifying bus transmission properties

Allowed_Connection_Type: list of enumeration
                        (Sampled_Data_Connection, Immediate_Data_Connection,
                         Delayed_Data_Connection, Port_Connection,
                         Data_Access_Connection,
                         Subprogram_Access_Connection)

Allowed_Physical_Access_Class: list of classifier ( device, processor, memory, bus )
Allowed_Physical_Access: list of reference ( device, processor, memory, bus )
Allowed_Message_Size: Size_Range

Transmission_Type: enumeration ( push, pull )

Transmission_Time: record (
    Fixed: Time_Range;
    PerByte: Time_Range; )

Prototype_Substitution_Rule: inherit enumeration (Classifier_Match, Type_Extension,
Signature_Match)

-- Hardware description properties

Hardware_Description_Source_Text: inherit list of aadlstring

Hardware_Source_Language: Supported_Hardware_Source_Languages

-- Data movement related properties

Assign_Time: record (
    Fixed: Time_Range;
    PerByte: Time_Range; )

Access_Right : Access_Rights => read_write

-- Protocol support

Provided_Virtual_Bus_Class : inherit list of classifier (virtual bus)
```

```
Provided_Connection_Quality_Of_Service : inherit list of Supported_Connection_QoS
-- mode related properties
Resumption_Policy: enumeration ( restart, resume )
-- Virtual machine layering
Implemented_As: classifier ( system implementation )
```

Semantics

(5) A bus support physical communication between processors, memories, and devices. This allows a processor to support execution of source text in the form of code and data loaded as binary images into memory components. A bus allows a processor to access device hardware when executing device software. A bus may also support different port and subprogram connections between thread components bound to different processors. The `Allowed_Connection_Type` property indicates which forms of access a particular bus supports. The bus may constrain the size of messages communicated through data or event data connections.

(6) A bus component provides access between processors, memories, and devices. It is a shared component, for which access is required by each of the respective components. A device is accessible from a processor if the two share a bus component. A memory is accessible from a processor if the two share a bus component.

(7) Buses can be directly connected to other buses. This is represented by one bus declaration specifying access to another bus in its requires subclause.

(8) Physical access to a bus can be restricted to certain types of devices, memory, buses, and processors. This is achieved with the property `Allowed_Physical_Access`.

(9) Bus components can have different property values under different operational modes.

(10) A bus component can include protocols in its abstraction. Protocols provided by a bus can be specified with the `Provided_Virtual_Bus_Class` property. They are matched against protocol requirements of connections and virtual buses as specified by their `Required_Virtual_Bus-Class` property. If desired instances of protocols supported by a bus can be explicitly modeled as virtual bus subcomponents. In that case the connection or virtual bus requiring a specific protocol can be bound to the specific virtual bus instance.

(11) Virtual buses (protocols) may be implemented in the bus hardware or in software. The virtual bus software executes on processors connected to the bus, whose bound threads communicate over connections that require the protocol.

(12) The bus is intended to be an abstraction of a physical bus or network. If it is desirable, the internals of the bus can be modeled by AADL as a system in its own right, i.e., an application system and an execution platform. This system implementation description can be associated with the bus component declaration by the `Implemented_As` property (see Section 14).

Processing Requirements and Permissions

(13) A method of implementation shall define how the final size of a transmission is determined for a specific connection. Implementation choices and restrictions such as packetization and header and trailer information are not defined in this standard.

(14) A method of implementation shall define the methods used for bus arbitration and scheduling. If desired this can be modeled by a notation of your choice and associated with a bus via property. Alternatively, it can be modeled through an AADL system model and associated with the bus through an `Implemented_As` property.

Examples

```
package Hardware
bus VME
end VME;

memory Memory_Card
features
   Card_Connector : requires bus access VME;
end Memory_Card;

processor PowerPC
features
   Card_Connector : requires bus access VME;
end PowerPC;

processor implementation PowerPC.Linux
end PowerPC.Linux;

system Dual_Processor end Dual_Processor;

system implementation Dual_Processor.PowerPC
subcomponents
   System_Bus: bus VME;
   Left: processor PowerPC.Linux;
   Right: processor PowerPC.Linux;
   Shared_Memory: memory Memory_Card;

connections
   bus access System_Bus <-> Left.Card_Connector;
   bus access System_Bus <-> Right.Card_Connector;
   bus access System_Bus <-> Shared_Memory.Card_Connector;
end Dual_Processor.PowerPC;
end Hardware;
```

6.5 Virtual Buses

(1) A virtual bus component represents logical bus abstraction such as a virtual channel or communication protocol.

(2) Virtual buses can be bound to buses, virtual buses, processors, virtual processors, devices, or memory. When bound to a bus, it may represent multiple protocols supported by the bus or a virtual channel with a subset of the bus bandwidth. When bound to a virtual bus it may represent a hierarchy of protocols or virtual channels. When bound to a processor it may represent protocols supported by a processor. When bound to a sequence of hardware components it may represent a virtual channel or flow across these components.

(3) Virtual buses can be subcomponents of processors, buses, and other virtual buses. This implies that they are bound to the processor, bus, or virtual bus they are contained in.

(4) Connections and virtual buses can indicate by property the protocols they require by identifying the appropriate virtual bus or bus classifier. A connection can also indicate the desired level of quality of service, which must be satisfied by the virtual bus or bus the connection is bound to. Similarly, hardware components can indicate by property the virtual buses they provide.

Legality Rules

Category	Type	Implementation
virtual bus	Features • none Flow specifications: no Modes: yes Properties: yes	Subcomponents: • virtual bus • abstract Subprogram calls: no Connections: no Flows: no Modes: yes Properties: yes

(L1) A virtual bus type can have property associations.

(L2) A virtual bus type must not contain flow specifications.

(L3) A virtual bus implementation can contain virtual bus subcomponent declarations.

(L4) A virtual bus implementation can contain a modes subclause and property associations.

(L5) A virtual bus implementation must not contain a connections subclause, flows subclause, or subprogram calls subclause.

Consistency Rules

(C1) In a fully deployed system virtual buses must be directly or indirectly bound to processors or buses that support these virtual buses, or they must be subcomponents of buses and processors.

Standard Properties

```
-- Properties specifying bus transmission properties

Allowed_Connection_Type: list of enumeration
                        (Sampled_Data_Connection, Immediate_Data_Connection,
                         Delayed_Data_Connection, Port_Connection,
                         Data_Access_Connection,
                         Subprogram_Access_Connection)

Allowed_Message_Size: Size_Range
```

```
Transmission_Type: enumeration ( push, pull )

Transmission_Time: record (

    Fixed: Time_Range;

    PerByte: Time_Range; )

Prototype_Substitution_Rule: inherit enumeration (Classifier_Match, Type_Extension,
Signature_Match)

-- Protocol support

Provided_Connection_Quality_Of_Service : inherit list of Supported_Connection_QoS

Provided_Virtual_Bus_Class : inherit list of classifier (virtual bus)

Required_Connection_Quality_Of_Service : inherit list of Supported_Connection_QoS

Allowed_Connection_Binding_Class:
   inherit list of classifier(processor, virtual processor, bus, virtual bus, device,
memory)

Allowed_Connection_Binding: inherit list of reference (processor, virtual processor, bus,
virtual bus, device, memory)

Actual_Connection_Binding: inherit list of reference (processor, virtual processor, bus,
virtual bus, device, memory)

Required_Virtual_Bus_Class : inherit list of classifier (virtual bus)

-- mode related properties

Resumption_Policy: enumeration ( restart, resume )

-- Virtual machine layering

Implemented_As: classifier ( system implementation )
```


Semantics

(5) A virtual bus represents a virtual channel or communication protocol. The `Provided_Virtual_Bus_Class` property is used to indicate that a processor or bus supports a protocol. Similarly, the `Required_Virtual_Bus_Class` property is used to indicate that a protocol is required by a connection or virtual bus.

(6) A virtual bus can be a subcomponent of a virtual bus, bus, virtual processor, processor, or system, it can be provided as indicated by the `Provided_Virtual_Bus_Class` property of a bus, virtual bus, virtual processor, or processor. In all cases, this indicates that the protocol represented by the virtual bus is supported on the bus or processor.

(7) If a virtual bus requires another virtual bus as expressed by the `Required_Virtual_Bus_Class` property, this required access is satisfied by binding the protocol to a processor or bus that provides this virtual bus as specified with the `Provided_Virtual_Bus_Class` property.

(8) A virtual bus can represent a portion of the bandwidth capacity of a bus. It can act as virtual channel that can make certain performance guarantees.

(9) The `Allowed_Connection_Type` property indicates which forms of access a particular virtual bus supports. The virtual bus may constrain the size of messages communicated through data or event data connections.

(10) Virtual bus components can have different property values under different operational modes.

(11) If it is desirable, the internals of the virtual bus can be modeled by AADL as an application system in its own right, e.g., as a sender thread interacting with a receiver thread. This system implementation description can be associated with the virtual bus component declaration by the `Implemented_As` property (see Section 14).

Processing Requirements and Permissions

(12) A method of implementation shall define how the final size of a transmission is determined for a specific connection. Implementation choices and restrictions such as packetization and header and trailer information are not defined in this standard.

(13) A method of implementation shall define the methods used for bus arbitration and scheduling. If desired this can be modeled by a notation of your choice and associated with a virtual bus via a property. Alternatively, it can be modeled through an AADL system model and associated with the bus through an `Implemented_As` property.

Examples

```
package Hardware
bus Ethernet
end Ethernet;

virtual bus IP_TCP
end IP_TCP;

virtual bus HTTP
properties
  Allowed_Bus_Binding_Class => virtual bus IP_TCP;
end HTTP;

processor PowerPC
end PowerPC;

processor implementation PowerPC.Linux
subcomponents
  IP_TCP: virtual bus IP_TCP;
end PowerPC.Linux;
end Hardware;
```


6.6 Devices

(1) A device component represents dedicated hardware within the system, entities in the external environment, or entities that interface with the external environment. A device may represent a physical entity or its (simulated) software equivalent. A device may exhibit simple or complex behaviors. Devices may internally have a processor, memory and software that can be modeled in a separate system declaration and associated with the device through the `Implemented_As` property. Devices are logically connected to application software components and physically connected to processors via buses. If the device has associated software such as device drivers that must reside in a memory and execute on a processor external to the device, this can be specified through appropriate property values for the device.

(2) A device interacts with both execution platform components and application software components. A device can have physical connections to processors via a bus. This models software executing on a processor accessing the physical device. A device also has logical connections to application software components. Those logical connections are represented by connection declarations between device ports and application software component ports. For any logical connection between a device and a thread executing application source text, there must be a physical connection in the execution platform.

(3) A device can be viewed to be a primary part of the application system. In this case, it is natural to place the device together with the application software components. The physical connection to processors must follow the system hierarchy.

(4) A device may be viewed to be primarily part of the execution platform. In this case, it is placed in proximity of other execution platform components. The logical connections have to follow the system hierarchy to connect to application software components.

(5) Examples of devices are sensors and actuators that interface with the external physical world, or standalone physical systems (such as a GPS) or dedicated hardware (such as counters or timers). Devices communicate with embedded application systems through ports and connections and with the computing hardware through bus access.

Legality Rules

Category	Type	Implementation
device	Features • port • feature group • provides subprogram access • provides subprogram group access • requires bus access • provides bus access • feature Flow specifications: yes Modes: yes Properties: yes	Subcomponents: • bus • virtual bus • abstract Subprogram calls: no Connections: yes Flows: yes Modes: yes Properties: yes

(L1) A device type can contain port, feature group, provides subprogram access, provides subprogram group access, bus access declarations, flow specifications, a modes subclause, as well as property associations.

(L2) A device component implementation must not contain a subprogram calls subclause.

(L3) A device implementation can contain abstract, virtual bus, and bus subcomponents, bus access connections, a modes subclause, a flows subclause, and property associations.

Standard Properties

```
-- Hardware description properties
Hardware_Description_Source_Text: inherit list of aadlstring
Hardware_Source_Language: Supported_Hardware_Source_Languages
-- Properties specifying device driver software that must be
-- executed by a processor
Source_Text: inherit list of aadlstring
Source_Language: inherit list of Supported_Source_Languages
Source_Code_Size: Size
Source_Data_Size: Size
```

```
Source_Stack_Size: Size
-- Properties specifying the execution properties of the device or its driver
Dispatch_Protocol: Supported_Dispatch_Protocols
Dispatch_Trigger: list of reference (port)
Period: inherit Time
Compute_Execution_Time: Time_Range
Deadline: inherit Time => Period
-- scheduling properties
Time_Slot: list of aadlinteger
Priority: inherit aadlinteger
-- Properties specifying constraints for processor and memory binding
-- for the device driver software
Allowed_Memory_Binding_Class:
   inherit list of classifier (memory, system, processor)
Allowed_Memory_Binding: inherit list of reference (memory, system, processor)
Actual_Memory_Binding: inherit list of reference (memory)
Actual_Processor_Binding: inherit list of reference (processor, virtual processor)
Allowed_Processor_Binding_Class:
   inherit list of classifier (processor, virtual processor, system)
Allowed_Processor_Binding: inherit list of reference (processor, virtual processor,
system)
-- protocol support
Provided_Virtual_Bus_Class : inherit list of classifier (virtual bus)
Provided_Connection_Quality_Of_Service : inherit list of Supported_Connection_QoS
-- mode related properties
Resumption_Policy: enumeration ( restart, resume )
-- Virtual machine layering
Implemented_As: classifier ( system implementation )
```


Semantics

(6) AADL device components model dedicated hardware or physical entities in the external environment, e.g., a GPS system, or entities that interface with an external environment, e.g., sensors and actuators as interface between a physical plant and a control system. Devices may represent a physical entity or its (simulated) software equivalent. They may exhibit simple behavior, e.g., a timer, or complex behaviors, e.g., a camera or GPS. Devices are logically connected to application software components and physically connected to processors. The device functionality may be fully embedded in the device hardware, or it may be provided by device-specific driver software.

(7) A device is accessible from a processor if the device is connected via a shared bus component.

(8) A device declaration can include flow specifications that indicate that a device is a flow source, a flow sink, or a flow path exists through a device.

(9) Device components can have different property values under different operational modes.

(10) Embedded application software interacts with devices through port connections and through subprogram calls. Data ports can be used to represent device registers. Event and event data ports can be used to represent signals and interrupts that trigger execution of software or that initiate a reaction by the device. Subprogram calls reflect functionality executed by the device. This functionality may be fully implemented in the device hardware or through a device driver.

(11) The `Dispatch_Protocol` property determines the execution behavior of the device. A device can execute periodically or event-driven. Periodic execution means that the device polls the external environment periodically to produce a periodic data stream to the application, or that it samples input from the application periodically. Event-driven execution means that the device generates events, e.g., a clock or timer, that it observes events in the external environment and passes them to the application, e.g., flipping of a switch, or that it responds to events or event data being sent by the application, e.g., a signal to turn on a light. The input or output rate on device ports can be specified through the `Output_Rate` property. The `Dispatch_Trigger` property can be used to specify a subset of event or event data ports that can trigger a device dispatch.

(12) The interface to a device may be through a device driver. The features of the device type may represent the interface to the device via the device driver. The execution behavior of the device driver is specified by the device dispatch protocol.

(13) The device driver may execute as part of the underlying operating system kernel. In this case, we can specify the driver characteristics as properties on the device itself, such as the code size, data size, and stack size. Binding properties specify the processor whose runtime system includes the driver.

(14) The device driver may execute in a separate address space from the operating system kernel. In this case, the binding property may specify a virtual processor as target.

(15) A device driver may execute as an application thread. In this case, the driver is modeled by an AADL thread. This thread provides the device interface to the application and interfaces with the device registers of the physical device, represented by the AADL device component.

(16) A device can contain a bus subcomponent that it makes externally accessible. This models a bus that is managed by the device and other components can connect to it. In this case the device is implicitly connected to this bus.

(17) A device can support communication protocols as expressed by the `Provided_Virtual_Bus_Class` property. Instances of such protocols can also be explicitly represented by virtual bus subcomponents.

(18) The device is intended to be an abstraction of a physical component in the embedded system environment. If it is desirable, the internals of the device can be modeled by AADL as a system in its own right, i.e., an application system and an execution platform. For example, a digital camera as a device can be modeled as a set of processors and application software that handles the taking of images and their transfer from the camera to a processor via a USB bus connection. This system implementation description can be associated with the device component declaration by the `Implemented_As` property (see Section 14).

Processing Requirements and Permissions

(19) This software must reside as a binary image on memory components and is executed on a processor component. The executing processor that has access to the device must be connected to the device via a bus. The memory storing the binary image must be accessible to the processor. Device driver software may be modeled implicitly through the `Source_Text` and related properties, or it may be modeled explicitly by a separate thread declaration.

(20) In the implicit model the execution of the device driver software may be considered to be part of the runtime system of a processor to which the device is connected, or it may be treated as an implicitly declared thread.

Examples

```
Package Equipment
device Camera
features
  usbport: requires bus access Buses::USB.USB2;
  image: out event data port UserTypes::imageformat.jpg;
end Camera;

device temperature_sensor
features
  serialline: requires bus access RS232;
  temp_reading: out data port UserTypes::Temperature.Celsius;
end temperature_sensor;

device implementation temperature_sensor.hardware
properties
  HardwareDescription_Source_Text => "TemperatureSensorHardwareModel.mdl";
end temperature_sensor.hardware;

device implementation temperature_sensor.simulation
properties
  Simulation::SensorReadings => "SensorTrace1.xls";
end temperature_sensor.simulation;

device Timer
features
  SignalWire: requires bus access Wire.Gauge12;
  SetTime: in event data port UserTypes::Time;
  TimeExpired: out event port;
end Timer;
end Equipment;
```

7 System Composition

(1) Systems are organized into a hierarchy of components to reflect the structure of actual systems being modeled. This hierarchy is modeled by *system* declarations to represent a composition of components into composite components. A *system instance* models an instance of an application system and its binding to a system that contains execution platform components.

7.1 Systems

(1) A system represents an assembly of interacting application software, execution platform, and system components. Systems can have multiple modes, each representing a possibly different configuration of components and their connectivity contained in the system. Systems may require access to data and bus components declared outside the system and may provide access to data and bus components declared within. Systems may be hierarchically nested.

Legality Rules

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

(L1) A system component type can contain subprogram, subprogram group, data and bus access declarations, port, feature group declarations. It can also contain flow specifications as well as property associations.

(L2) A system component implementation can contain abstract, data, subprogram, subprogram group, process, and system subcomponent declarations as well as execution platform components, i.e., processor, virtual processor, memory, bus, virtual bus, and device.

(L3) A system implementation can contain a modes subclause, a connections subclause, a flows subclause, and property associations.

Standard Properties

```
-- Properties related to source text
Source_Text: inherit list of aadlstring
Source_Language: inherit list of Supported_Source_Languages
-- Process property that can be specified at the system level as well
-- Runtime enforcement of address space boundaries
```

```
Runtime_Protection : inherit aadlboolean

-- Inheritable thread properties

Synchronized_Component: inherit aadlboolean => true

Active_Thread_Handling_Protocol:
   inherit Supported_Active_Thread_Handling_Protocols => abort
Period: inherit Time

Deadline: inherit Time => Period

-- execution time related properties

Reference_Processor: inherit classifier ( processor )

-- scheduling properties

Time_Slot: list of aadlinteger

Priority: inherit aadlinteger

-- Properties related binding of software component source text in

-- systems to processors and memory

Allowed_Processor_Binding_Class:
   inherit list of classifier (processor, virtual processor, system)

Allowed_Processor_Binding: inherit list of reference (processor, virtual processor,
system)

Actual_Processor_Binding: inherit list of reference (processor, virtual processor)

Allowed_Memory_Binding_Class:
   inherit list of classifier (memory, system, processor)

Allowed_Memory_Binding: inherit list of reference (memory, system, processor)


Actual_Memory_Binding: inherit list of reference (memory)

Allowed_Connection_Binding_Class:
   inherit list of classifier(processor, virtual processor, bus, virtual bus, device,
memory)

Allowed_Connection_Binding: inherit list of reference (processor, virtual processor, bus,
virtual bus, device, memory)

Actual_Connection_Binding: inherit list of reference (processor, virtual processor, bus,
virtual bus, device, memory)

Collocated: record (

    Targets: list of reference (data, thread, process, system, connection);

    Location: classifier ( processor, memory, bus, system ); )

Not_Collocated: record (

    Targets: list of reference (data, thread, process, system, connection);

    Location: classifier ( processor, memory, bus, system ); )

-- Properties related systems as execution platforms

Hardware_Source_Language: Supported_Hardware_Source_Languages

-- mode related properties

Resumption_Policy: enumeration ( restart, resume )

-- Properties related to startup of processor contained in a system

Startup_Deadline: Time
```

```
Startup_Execution_Time: Time_Range
-- Properties related to system load times
Load_Time: Time_Range
Load_Deadline: Time
-- Properties related to the hardware clock
Clock_Jitter: Time
Clock_Period: Time
Clock_Period_Range: Time_Range
```

Semantics

(2) A system component represents an assembly of software and execution platform components. All subcomponents of a system are considered to be contained in that system.

(3) Some system components consist of purely software components all of which must be bound to execution platform components outside the system itself. An example is an application software system. Some system components consist purely of computing hardware components. They represent aggregations of processor, memory, and bus components that act as the hardware platform. Some system component is a composition of devices and buses that represent the physical environment that the embedded software system interacts with. Some system components may be combinations of the above. Some system components are self-contained in that all contained software components are bound to execution platform components contained within the same system. Such self-contained systems may have external connectivity in the form of logical connection points represented by ports and physical connection points in the form of required or provided bus access. Examples, of such systems are database servers, GPS receivers, and digital cameras. Such self-contained systems with an external interface may represent the implementation of devices. The device representation takes a black-box perspective, while the system representation takes a white-box perspective and is associated with the device through the `Implemented_As` property.

(4) A system component can contain a modes subclause. Each mode can represent an alternative system configuration of contained subcomponents and their connections. The transition between modes is determined by the mode transition declarations of specific property associations.

Processing Requirements and Permissions

(5) Processing methods may restrict data, subprogram, and subprogram group subcomponents to be part of only one process address space. In that case they may require those subcomponents to be placed inside a process, thread group, or thread, and not be allowed in system implementations.

8 Features and Shared Access

(1) A *feature* is a part of a component type definition that specifies how that component interfaces with other components in the system.

(2) *Feature groups* represent groups of component features. Feature groups can contain feature groups. Feature groups can be used anywhere features can be used. Within a component, the features of a feature group can be connected to individually. Outside a component, feature groups can be connected as a single unit. Feature groups can be partially defined without specifying the elements to play the role of an abstract feature. It can later be refined into a group of one or more features.

(3) *Port* features represent a communication interface for the exchange of data and events between components. Ports are classified into data ports, event ports, and event data ports.

(4) *Subprogram access* features represent access to a subprogram to be called from other components, and the need for a component to call a subprogram instance locally, i.e., a subprogram that is declared in the containing process or one of its subcomponents, or to call a subprogram remotely, i.e., a subprogram instance in another process. When used in data components subprogram access features represent subprograms (methods) through which the data component is manipulated. When used in subprogram groups subprogram access features represent the subprograms of a subprogram library.

(5) *Subprogram group access* features represent sharing and required access to a subprogram library.

(6) *Parameter* features represent data values that can be passed into and out of subprograms. Parameters are typed with a data classifier reference.

(7) *Data access* represents communication via shared access to data components. A data component declared inside a component implementation is specified to be accessible to components outside using a provides access feature declaration. A component may indicate that it requires access to a data subcomponent declared outside utilizing a requires access feature declaration.

(8) *Bus access* represents physical connectivity of processors, memory, devices, and buses through buses. A bus component declared inside a component implementation is specified to be accessible to components outside using a provides access feature declaration. A component may indicate that it requires access to a bus utilizing a requires access feature declaration.

(9) Features can be declared as one-dimensional feature arrays. Such feature arrays complement component arrays and allow for connection patterns that connect a feature of each of the component array elements to a feature array element of one component. An example use is redundant replicas of a component passing their output to a voting component or a routing component.

Syntax

```
feature ::=

    ( abstract_feature_spec |
      port_spec |
      feature_group_spec |
      subcomponent_access |
     parameter )
    [ array_dimension ]
    [ { { feature_property_association }+ } ] ;


subcomponent_access ::=
    subprogram_access | subprogram_group_access
```

```
    | data_access | bus_access

feature_refinement ::=
    abstract_feature_spec_refinement
    port_refinement |
    feature_group_refinement |
    subcomponent_access_refinement |
    parameter_refinement
    [ array_dimension ]
    [ { { feature_property_association }+ } ] ;

subcomponent_access_refinement ::=
    subprogram_access_refinement | subprogram_group_access_refinement
    | data_access_refinement | bus_access_refinement
```

Naming Rules

(N1) The defining identifier of a feature must be unique within the namespace of the associated component type.

(N2) Thread features may not be declared using the predeclared ports names `Complete` or `Error`.

(N3) Each refining feature identifier that appears in a feature refinement declaration must also appear in a feature declaration of a component type being extended.

(N4) A feature is referenced in one of two ways. Within the component implementations for a component type, a feature declared in the type is named in the implementations by its identifier. Within component implementations that contain subcomponents with features, a subcomponent feature is named by the subcomponent identifier and the feature identifier separated by a "." (dot).

Legality Rules

(L1) Each feature can be refined at most once in the same type extension.

(L2) A feature refinement declaration of a feature and the original feature must both be declared as port, parameter, access feature, or feature group, or the original feature must be declared as abstract feature.

(L3) Feature arrays must only be declared for threads, devices, and processors.

(L4) If the feature refinement specifies an array dimension, then the feature being refined must have an array dimension.

(L5) If the refinement specifies an array dimension size, then the feature being refined must not have an array dimension size.

Standard Properties

```
Acceptable_Array_Size: list of Size_Range
Required_Connection_Quality_Of_Service : inherit list of Supported_Connection_QoS
Required_Virtual_Bus_Class : inherit list of classifier (virtual bus)
```

Semantics

(10) A feature declaration specifies an interaction point with other components. Features can be connected to features of other components external to the component, and they can also be connected to subcomponents within component implementations associated with the component type with the feature declaration.

(11) A refined feature declaration may complete an incomplete component classifier reference and declare feature property associations. A feature refinement may replace a classifier reference according to the `Classifier_Substitution_Rule` property.

(12) Features can be declared with an array dimension, if the component is a thread, device, or processor. In this case, each element of the feature array is connected to a different element of an ultimate source or destination component array.

(13) For example, we may have a voting component that takes input from multiple instances of the same processing component, as shown in Figure 11. We can declare the processing component as an array of subcomponents, and the single instance of the voting component with a port array. We can then declare a connection from the outgoing port of the processing subcomponent to the port of the voting component declared with array dimensions. The connection can have a `Connection_Pattern` property of `One_To_One` (see Section 9.2.3) to indicate that each processing component output is connected to a separate port of the voting component. This is illustrated in the example below.

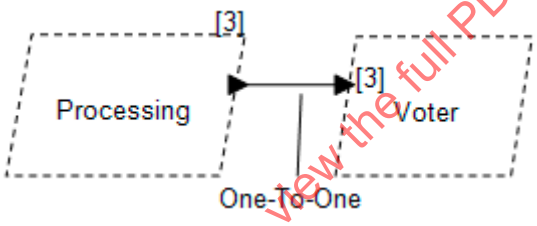


Figure 11 Port Array in a Voting Pattern

(14) A feature may specify desired quality of service or a particular protocol to be used for connections through the feature. This property must be consistent with the same property associated with the connection.

8.1 Abstract Features

(1) Abstract features represent placeholders for concrete features, i.e., ports, parameters, and the different kinds of access features. Abstract features are typically used in incomplete component type declarations, especially those that play the role of a template. They can be used in conjunction with prototypes and allow users to refine such features to concrete features.

Syntax

```
abstract_feature_spec ::=
     ( [  in | out ] feature
       [ unique_component_classifier_reference
       | component_prototype_identifier ] )
     | ( [  in | out ] feature feature_prototype_identifier )

abstract_feature_spec_refinement ::= abstract_feature_spec
```

Legality Rules

(L1) The feature direction in a refined feature declaration must be identical to the feature direction in the feature declaration being refined, or the feature being refined must not have a direction.

(L2) If the direction of an abstract feature is specified, then the direction must be satisfied by the refinement (see also the rules for feature prototypes in Section 4.7); in the case of ports the direction must be **in** or **out**; in the case of data access, the access right must be read-only for **in** and write-only for **out**; in the case of bus access, subprogram access and subprogram group access the direction must not be **in** nor **out**.

(L3) An abstract feature with a feature prototype identifier and the prototype being referenced must both specify the same direction or no direction.

(L4) In the case of an abstract feature with a classifier reference, the classifier of the refined feature declaration in a component type extension must adhere to the classifier refinement rules as indicated by the `Classifier_Substitution_Rule` property (see Section 4.5). By default, the `Classifier_Match` rule applies, i.e., an incomplete classifier reference can be completed.

(L5) An abstract feature refinement declaration of a feature with a feature prototype reference must only add property associations.

Semantics

(2) A component type can contain an abstract feature declaration that can later be refined into a feature group, port feature, access feature, or parameter. An abstract feature may be specified with a direction; this direction must be satisfied by any refinement according to legality rules.

(3) A component type can contain an abstract feature declaration with a feature prototype reference. In that case it is a placeholder for the feature being supplied by the prototype binding. A component type can contain an abstract feature declaration with a component prototype reference. In that case it declares a feature with the prototype binding supplying the component classifier.

8.2 Feature Groups and Feature Group Types

(1) Feature groups represent groups of component features or feature groups. Within a component, the features of a feature group can be connected to individually. Outside a component, feature groups can be connected as a single unit. This grouping concept allows the number of connection declarations to be reduced, especially at higher levels of a system when a number of features from one subcomponent and its contained subcomponents must be connected to features in another subcomponent and its contained subcomponents. The content of a feature group is declared through a feature group type declaration. This declaration is then referenced when feature groups are declared as component features. Feature groups can be declared for any kind of feature, for ports, and for access features.

Syntax

```
-- Defining the content structure of a feature group
feature_group_type ::=
    feature group defining_identifier
    [ prototypes ( { prototype }+ | none_statement ) ]
    [ features { feature }+ ]
    [ inverse of unique_feature_group_type_reference ]
    [ properties ( { feature_group_property_association }+ | none_statement ) ]
    { annex_subclause }*
end defining_identifier ;

feature_group_type_extension ::=
    feature group defining_identifier
    extends unique_feature_group_type_reference [ prototype_bindings ]
```

```
    [ prototypes ( { prototype }+ | none_statement ) ]
    [ features { feature | feature_refinement }+ ]
    [ inverse of unique_feature_group_type_reference ]
[ properties ( { feature_group_property_association }+ | none_statement ) ]
 { annex_subclause }*
end defining_identifier ;

-- declaring a feature group as component feature
feature_group_spec ::=
  defining_feature_group_identifier : [ in | out ] feature group
      [ [ inverse of ]
        ( unique_feature_group_type_reference | prototype_identifier ) ]

feature_group_refinement ::=
  defining_feature_group_identifier : refined to [ in | out ] feature group
        [ [ inverse of ]
          ( unique_feature_group_type_reference) | prototype_identifier ) ]

unique_feature_group_type_reference ::=
    [ package_name :: ] feature_group_type_identifier
```

Naming Rules

(N1) The defining identifier of a feature group type must be unique within the package namespace of the package where the feature group type is declared.

(N2) Each feature group type provides a local namespace. The defining identifiers of prototype, feature, and feature group declarations in a feature group type must be unique within the namespace of the feature group type.

(N3) The local namespace of a feature group type extension includes the defining identifiers in the local namespace of the feature group type being extended. This means, the defining identifiers of prototype, feature, or feature group declarations in a feature group type extension must not exist in the local namespace of the feature group type being extended. The defining identifiers of prototype, feature, or feature group refinements in a feature group type extension must refer to a prototype, feature, or feature group in the local namespace of an ancestor feature group type.

(N4) The defining feature identifiers of feature group declarations must be unique in the local name space of the component type containing the feature group declaration.

(N5) The defining feature group identifier of `feature_refinement` declarations in component types must exist in the local namespace of the component type being extended and must refer to a feature or feature group.

(N6) The package name of the unique feature group type reference must refer to a package name in the global namespace. The feature group type identifier of the unique feature group type reference must refer to a feature group type identifier in the named package.

(N7) The prototype reference in a feature group declaration must refer to a prototype of the component type or feature group type that contains the feature group declaration.

Legality Rules

(L1) A feature group type may contain zero or more elements, i.e., feature or feature groups. If it contains zero elements, then the feature group type may be declared to be the inverse of another feature group type.

(L2) A feature group type can be declared to be the inverse of another feature group type, as indicated by the reserved words **inverse of** and the name of a feature group type. Any feature group type named in an **inverse of** statement cannot itself contain an **inverse of** statement. This means that several feature groups can be declared to be the inverse of one feature group, e.g., `B inverse of A` and `C inverse of A` is acceptable. However, chaining of inverses is not permitted, e.g., `B inverse of A` and `C inverse of B` is not acceptable.

(L3) Only feature group types without **inverse of** or feature group types with features and **inverse of** can be extended.

(L4) A feature group type that is an extension of another feature group type without an **inverse of** cannot contain an **inverse of** statement.

(L5) The feature group type that is an extension of another feature group type with features and **inverse of** that adds features must have an **inverse of** to identify the feature group type whose inverse it is.

(L6) A feature group declaration with an **inverse of** statement must only reference feature group types without an **inverse of** statement.

(L7) A feature group refinement may be refined to only add property associations. In this case inclusion of the feature group type reference is optional.

Two feature group types are considered to complement each other if the following holds:

(L8) The number of feature or feature groups contained in the feature group and its complement must be identical;

(L9) Each of the declared features or feature groups in a feature group must be a pair-wise complement with that in the feature group complement, with pairs determined by declaration order. In the case of feature group type extensions, the feature and feature group declarations in the extension are considered to be declared after the declarations in the feature group type being extended;

(L10) If both feature group types have zero features, then they are considered to complement each other;

(L11) Ports are pair-wise complementary if they satisfy the port connection rules specified in Section 9.2.1. This includes appropriate port direction and matching of data component classifier references according to classifier matching rules (see Section 9.5 legality rules (L3) and (L4);

(L12) Access features are pair-wise complementary if they satisfy the access connection rules in Section 9.4.

(L13) If an **in** or **out** direction is specified as part of a feature group declaration, then all features inside the feature group must satisfy this direction.

NOTES:

Aggregate data ports can be modeled in AADL V2 by a data port with a data component classifier that has data subcomponents for each of the element ports. This replaces the `Aggregate_Data_Port` on port groups in the original AADL standard.

Standard Properties

```
-- Port properties defined to be inherit, thus can be associated with a
-- feature group to apply to all contained ports.
Source_Text: inherit list of aadlstring
Allowed_Memory_Binding_Class:
   inherit list of classifier (memory, system, processor)
```

```
Allowed_Memory_Binding: inherit list of reference (memory, system, processor)
Actual_Memory_Binding: inherit list of reference (memory)
```

Semantics

(2) A feature group declaration represents groups of component features. As such each feature group of a component type can represent a separate interface to the component.

(3) A feature group of a component can be connected to another component through a single connection declaration. It represents a connection for each of the feature inside the feature group. Feature groups can contain feature groups. This supports nested grouping of features for different levels of the modeled system.

(4) Within a component, the features of a feature group can be connected to individually to subcomponents. The members of the feature group are declared in a feature group type declaration that is referenced by the feature group declaration. The referenced feature group type determines the feature group compatibility for a feature group connection.

(5) The **inverse of** reserved words of a feature group type declaration indicate that the feature group type represents the complement to the referenced feature group type. The legality of feature group connections is affected by the complementary nature of feature groups (see Section 9.5).

(6) Features can be declared without feature group types or with feature group types without features. They are considered to be incomplete feature group specifications. Feature group types can later be added in a feature group refinement. Features can later be inserted directly into the feature group type or the feature group type can later be refined into feature group types with one or more features.

Examples

```
package GPS_Interface
feature group GPSbasic_socket
features
   Wakeup: in event port;
   Observation: out data port GPSLib::position;
end GPSbasic_socket;

feature group GPSbasic_plug
features
   WakeupEvent: out event port;
   ObservationData: in data port GPSLib::position;
   -- the features must match in same order with opposite direction
   inverse of GPSbasic_socket
end GPSbasic_plug;

feature group MyGPS_plug
   -- second feature group as inverse of the original
   -- no chaining in inverse and
   -- no pairwise inverse references are allowed
   inverse of GPSbasic_socket
end MyGPS_plug;
```

```
feature group GPSextended_socket extends GPSbasic_socket
features
   Signal: out event port;
   Cmd: in data port GPSLib::commands;
end GPSextended_socket;

process Satellite_position
features
   position:   feature group GPSBasic_socket;
end Satellite_position;

process GPS_System
features
   position: feature group inverse of GPSbasic_socket;
end GPS_System;

system implementation Satellite.others
subcomponents
   SatPos: process Satellite_position;
   MyGPS: process GPS_System;
connections
   feature group Satpos.position -> MyGPS.position;
end Satellite.others;
end GPS_Interface;
```


8.3 Ports

(1) Ports are logical connection points between components that can be used for the transfer of control and data between threads or between a thread and a processor or device. Ports are directional, i.e., an output port is connected to an input port. Ports can pass data, events, or both. Data transferred through ports is typed. From the perspective of the application source text, data ports are accessible in the source text as data variables. From the perspective of the application source text, event ports represent event queues whose size is accessible. Incoming events may trigger thread dispatches or mode transitions, or they may simply be queued for processing by the recipient. From the perspective of the application source text, event data ports represent message queues whose content can be retrieved.

(2) The content of incoming ports are frozen at a specified time, by default at dispatch time. This means that the content of the port that is accessible to the recipient does not change during the execution of a dispatch even though the sender may send new values. Properties specify the input and output timing characteristics of ports. Actual event and data transfer may be initiated by the runtime system of the execution platform or by `Send_Output` runtime service calls in the application source text.

(3) AADL distinguishes between three port categories. *Event data ports* are ports through which data is sent and received. The arrival of data at the destination may trigger a dispatch or a mode switch. The data may be queued if the destination component is busy. Event data ports effectively represent message ports. *Data ports* are event data ports with a queue size of one in which the newest arrival is kept. By default arrival of data at data ports does not trigger a dispatch. Data ports effectively represent unqueued ports that communicate state information, such as signal streams that are sampled and processed in control loops. *Event ports* are event data ports with empty message content. Event ports effectively represent discrete events in the physical environment, such as a button push, in the computing platform, such as a clock interrupt, or a logical discrete event, such as an alarm.

Syntax

```
port_spec ::=
    defining_port_identifier : ( in | out | in out ) port_type

port_refinement ::=
    defining_port_identifier : refined to
        ( in | out | in out ) port_type

port_type ::=
     data port [ data_unique_component_classifier_reference
               | data_component_prototype_identifier ]
    | event data port [ data_unique_component_classifier_reference
                      | data_component_prototype_identifier ]
    | event port
```

Naming Rules

(N1) A defining port identifier must adhere to the naming rules specified for all features (see Section 8).

(N2) The defining identifier of a port refinement declaration must also appear in a feature declaration of a component type being extended and must refer to a port or an abstract feature.

(N3) The unique component type identifier of the data classifier reference must be the name of a data component type. The data implementation identifier, if specified, must be the name of a data component implementation associated with the data component type.

(N4) The prototype identifier of a prototype reference, if specified, must exist in the namespace of the component type or feature group type that contains the feature declaration.

Legality Rules

(L1) Ports can be declared in subprogram, thread, thread group, process, system, processor, virtual processor, and device component types.

(L2) Data and event data ports may be incompletely defined by not specifying the data component classifier reference or data component implementation identifier of a data component classifier reference. The port definition can be completed using refinement.

(L3) Data, event, and event data ports may be refined by adding a property association. The data component classifier declared as part of the data or event data port declaration being refined does not need to be included in this refinement.

(L4) The port category of a port refinement must be the same as the category of the port being refined, or the port being refined must be an abstract feature.

(L5) The port direction of a port refinement must be the same as the direction of the feature being refined. If the feature being refined is an abstract feature without direction, then all port directions are acceptable.

Standard Properties

```
-- Properties specifying the source text variable representing the port
Source_Name: aadlstring
Source_Text: inherit list of aadlstring
-- property indicating whether port connections are required or optional
Required_Connection : aadlboolean => true


-- The protocol the source text supporting the port is assumed to make use of
Allowed_Connection_Binding_Class:
   inherit list of classifier(processor, virtual processor, bus, virtual bus, device,
memory)
-- Optional property for device ports
Device_Register_Address: aadlinteger
-- data port connection timing
Timing : enumeration (sampled, immediate, delayed) => sampled
-- Input and output rate and time
Input_Rate: Rate_Spec => ( Value_Range => 1.0 .. 1.0; Rate_Unit => PerDispatch;
Rate_Distribution => Fixed; )
Input_Time: list of IO_Time_Spec => ( Time => Dispatch; Offset => 0.0 ns .. 0.0 ns;)
Output_Rate: Rate_Spec => ( Value_Range => 1.0 .. 1.0; Rate_Unit => PerDispatch;
Rate_Distribution => Fixed; )
Output_Time: list of IO_Time_Spec => ( Time => Completion; Offset => 0.0 ns .. 0.0 ns;)
-- Port specific compute entrypoint properties for event and event data ports
Compute_Entrypoint: classifier ( subprogram classifier )
Compute_Execution_Time: Time_Range
Compute_Deadline: Time
-- Properties specifying binding constraints for variables representing ports
Allowed_Memory_Binding_Class:
   inherit list of classifier (memory, system, processor)
Allowed_Memory_Binding: inherit list of reference (memory, system, processor)
Actual_Memory_Binding: inherit list of reference (memory)
-- In port queue properties
Overflow_Handling_Protocol: enumeration (DropOldest, DropNewest, Error)
       => DropOldest
Queue_Size: aadlinteger 0 .. Max_Queue_Size => 1
Queue_Processing_Protocol: Supported_Queue_Processing_Protocols => FIFO
Fan_Out_Policy: enumeration (Broadcast, RoundRobin, Selective, OnDemand)
```

```
Urgency: aadlinteger 0 .. Max_Urgency
Dequeued_Items: aadlinteger
Dequeue_Protocol: enumeration ( OneItem, MultipleItems, AllItems ) => OneItem
```

Semantics

8.3.1 Port Categories

(4) A port specifies a logical connection point in the interface of a component through which incoming or outgoing data and events may be passed. Ports may be named in connection declarations. Ports that pass data are typed by naming a data component classifier reference.

(5) A data or event data port maps to a static variable in the source text that represents the data buffer or queue. By default the variable is accessible by the same name as the port name. A different name mapping can be specified with the `Source_Name` and `Source_Text` properties. The `Allowed_Memory_Binding` and `Allowed_Memory_Binding_Class` properties indicate the memory (or device) hardware the port resources reside on.

(6) Event and event data ports may dispatch a port specific `Compute_Entrypoint`. This permits threads with multiple event or event data ports to execute different source text sequences for events arriving at different event ports. If specified, the port specific `Compute_Execution_Time` and `Compute_Deadline` takes precedence over those of the containing thread.

(7) Ports are directional. An **out** port represents output provided by the sender, and an **in** port represents input needed by the receiver. An **in out** port represents both an **in** port and an **out** port. Incoming connection(s) and outgoing connection(s) of an **in out** port may be connected to the same component or to different components. An **in out** port maps to a port variable in the source text. This means that the source text will overwrite the existing incoming value of the port when writing the output value to the port variable.

(8) Ports that provide output, i.e., **out** ports or **in out** ports, are referred to as outgoing port. Ports that provide input, i.e., **in** ports or **in out** ports, are referred to as incoming ports.

(9) A port can require a connection or consider it as optional as indicated by the `Required_Connection` property. In the latter case it is assumed that the component with this port can function without the port being connected.

(10) Ports appear to the thread as input and output buffers, accessible in source text as port variables.

(11) Data and event data ports are used to transmit data between threads.

(12) Data ports are intended for transmission of state data such as sensor data streams. Therefore, no queuing is supported for data ports. A thread can determine whether the input buffer of an in data port has new data at this dispatch by checking the port status through a `Get_Count` service call, which is accessible through the port variable through a `Get_Value` service call. If no new data value has been received the old value is made available.

(13) Event data ports are intended for message transmission, i.e., the queuing of the event and associated data at the port of the receiving thread. A receiving thread can get access to one or more data element in the queue according to the `Dequeue_Protocol` and `Dequeued_Items` properties (see Section 8.3.3). The number of queued event data elements accessible to a thread can be determined through the port variable using the `Get_Count` service call. Individual element of the queue can be retrieved via the port variable using the `Get_Value` and `Next_Value` service calls. If the queue is empty the most recent data value is available.

(14) Event ports are intended for event and alarm transmission, i.e., the queuing of events at the port of the receiving thread, possibly resulting in a dispatch or mode transition. A receiving thread can get access to one or more events in the queue according to the `Dequeue_Protocol` and the `Dequeue_Items` property. The number of queued events accessible to a thread can be determined through the port variable by making a `Get_Count` service call.

(15) The role of an aggregate data port is to make a collection of data from multiple outgoing data ports available in a time-consistent manner. Time consistency in this context means that if a set of periodic threads is dispatched at the same time to operate on data, then the recipients of their data see either all old values or all new values. This is accomplished by declaring a data port, whose data classifier has an implementation with data components corresponding to the data of the individual data ports. The functionality of an aggregate data port can be viewed as a thread whose only role is to collect the data values from several **in** data ports and make them available as an aggregate data record; on the receiving side an equivalent thread takes passes on the elements of the aggregate data record on to the respective **out** data ports of receiving threads. The function may be optimized by mapping the data ports of the individual threads into a data area representing the aggregate data port variable. This aggregate is then transferred as a single unit.

8.3.2 Port Input and Output Timing

(16) Data, events, and event data arriving through incoming ports is made available to the receiving thread, processor, or device at a specified input time. For a data port the input that is available through a port variable is a data value, while for an event or event data port it can be one or more elements from the port queue according to a specified dequeuing protocol (see Section 8.3.3). From that point on any newly arriving data, event, or event data is not available to the receiving component until the next dispatch, i.e., the content of an incoming port that is accessible to the application code does not change while the thread completes its execution.

(17) By default, port input is frozen at dispatch time. For periodic threads or devices this means that input is sampled at fixed time intervals.

(18) The `Input_Time` property can be used to explicitly specify an input time for ports. This can be done for all ports by specifying the property value for the thread, or it can be specified separately for each port.

(19) The following property values for `Input_Time` are supported to specify the input time to be the dispatch time (`Dispatch`), any time during execution relative to the amount of execution time from the start (`Start`) or from the completion (`Completion`), and the fact that no input occurs (`NoIO`):

- `Dispatch_Time`: (the default value) input is frozen at dispatch time; the time reference is clock time $t = 0$.
- `Start`, time range: input is frozen at a specified amount of execution time from the beginning of execution. The time is within the specified time range. The time range must have positive values. $Start_{low} \leq c \leq Start_{high}$.
- `Completion`, time range: input is frozen at a specified amount of execution time relative to execution completion. The time is within the specified time range. A negative time range indicates execution time before completion. $c_{complete} + Completion_{low} \leq c \leq c_{complete} + Completion_{high}$, where $c_{complete}$ represents the value of c at completion time.
- `NoIO`: input is not frozen. In other words, the port is excluded from making new input available to the source program. This allows users to specify that a subset of ports to provide input. The property value can be mode specific, i.e., a port can be excluded in one mode and included in another mode.

(20) The `Input_Time` property can have a list of values. In this case it indicates that input is frozen multiple times for the execution of a dispatch. If a thread has multiple input times specified, then the content of an incoming port is frozen multiple times during a single dispatch.

(21) The input may be frozen at dispatch time (`Input_Time` property value of `Dispatch`) as part of the underlying runtime system, or it may be frozen through a `Receive_Input` service call in the source text (`Input_Time` property value of `Start` or `Completion`).

(22) The input of other ports that can trigger dispatch is not frozen. Input of the remaining ports is frozen according to the specified input time.

(23) If a dispatch condition is specified then the logic expression determines the combination of event and event data ports that trigger a dispatch, and whose input is frozen as part of the dispatch. The input of other ports that can trigger dispatch is not frozen. The input of other ports that can trigger dispatch is not frozen. Input of the remaining ports is frozen according to the specified input time.

(24) If an event port is associated with a component (including thread) containing modes and mode transition, and the mode transition names the event port, then the arrival of an event is a mode change request and it is processed according to the mode switch semantics (see Sections 12 and 13.6).

(25) By default, the output time, i.e., the time output is transmitted to connected components, is the completion time for data ports. By default, for event and event data ports the output time occurs anytime during the execution through a `Send_Output` service call.

(26) The `Output_Time` property can be used to explicitly specify an output time for ports. This can be done for all ports by specifying the property value for the thread, or it can be specified separately for each port.

(27) The following property values for `Output_Time` are supported to specify the output time to be the dispatch time (`Dispatch`), any time during execution relative to the amount of execution time from the start (`Start`) or from the completion (`Completion`) including at completion time, the deadline (`Deadline`), and the fact that no input occurs (`NoIO`):

- `Start`, time range: output is transmitted at a specified amount of execution time relative to the beginning of execution. The time is within the specified time range. The time range must have positive values. $\texttt{Start}_{low} \leq c \leq \texttt{Start}_{high}$.
- `Completion`, time range: output is transmitted at a specified amount of execution time relative to execution completion. The time is within the specified time range. A negative time range indicates execution time before completion. $c_{complete} + \texttt{Completion}_{low} \leq c \leq c_{complete} + \texttt{Completion}_{high}$, where $c_{complete}$ repesents the value of c at completion time. The default is completion time with a time range of zero, i.e., it occurs at $c = c_{complete}$.
- `Deadline`: output is transmitted at deadline time; the time reference is clock time rather than execution time. `t = Deadline`. This allows for static alignment of output time of one thread with the `Dispatch_Time` input time of another thread with a `Dispatch_Offset`.
- `NoIO`: output is not transmitted. In other words, the port is excluded from transmitting new output from the source text. This allows users to specify that a subset of ports to provide output. The property value can be mode specific, i.e., a port can be excluded in one mode and included in another mode.

(28) The `Output_Time` property can have a list of values. In this case it indicates that output is transmitted multiple times as part of the execution of a dispatch.

(29) The output may be transmitted at completion time or deadline as part of the underlying runtime system, or it may be transmitted through a `Send_Output` service call in the source text.

(30) If the output time of the originating port is `Completion_Time` while the input time of the receiving port is `Dispatch` and the sender and receiver are in the same synchronization domain, then the output is received at the next dispatch equal to or later than the deadline. To accommodate the transfer the actual transfer may be initiated before the deadline. In the case of the connection crossing synchronization domains, the input is received at the dispatch following the completion of the transfer.

(31) The `Input_Rate` and `Output_Rate` properties specify the number of times per dispatch (`perDispatch`) or per second (`perSecond`) at which input and output is expected to occur at the port with the associated property. By default the input and output rate of ports is once per dispatch. The rate can be `fixed` or according to a distribution.

(32) An input or output rate higher than once per dispatch indicates that multiple inputs or multiple outputs are expected during a single dispatch. An input or output rate lower than once per dispatch indicates that inputs or outputs are not expected at every dispatch.

(33) If an `Input_Time` or `Output_Time` property is specified, then the values must be consistent with the rate. If the rate is specified in terms of seconds and a period is specified for the thread or device with the port, then the period value must also be consistent with the other values. In the case of an `Input_Time` or `Output_Time` property value of `None` the rate value does not apply.

Examples

```
-- a thread that gets input partway into execution and sends output
-- before completion.
thread TightLoop
features
   sensor: in data port
      {Input_Time => (Start,10 us .. 15 us);} ;
   actuator: out data port _
      {Output_Time => (Completion,10 us .. 11 us);} ;
end TightLoop;
```

8.3.3 Port Queue Processing

(34) Event and event data ports can have a queue associated with them. By default, the incoming event ports and event data ports of threads, devices, and processors have queues. The output from the ultimate source of a semantic port connection is added into this queue, if the ultimate destination component is actively processing. The default port queue size is 1 and can be changed by explicitly declaring a `Queue_Size` property association for the port.

(35) The `Queue_Size`, `Queue_Processing_Protocol`, and `Overflow_Handling_Protocol` port properties specify queue characteristics. If an event arrives and the number of queued events (and any associated data) is equal to the specified queue size, then the `Overflow_Handling_Protocol` property determines the action. If the `Overflow_Handling_Protocol` property value is `Error`, then an error occurs for the thread. The thread can determine the port that caused the error by calling the standard `Dispatch_Status` runtime service. For `Overflow_Handling_Protocol` property values of `DropNewest` and `DropOldest`, the newly arrived or oldest event in the queue event is dropped.

(36) Queues will be serviced according to the `Queue_Processing_Protocol`, by default in a first-in, first-out order (FIFO). When an aperiodic, sporadic, timed, or hybrid thread declares multiple in event and event data ports in its type that can be dispatch triggers and more than one of these queues are nonempty, the port with the higher `Urgency` property value gets serviced first. If several ports with the same `Urgency` are non-empty, then the `Queue_Processing_Protocol` is applied across these ports and must be the same for them. In the case of FIFO the oldest event will be serviced (global FIFO). It is permitted to define and use other algorithms for picking among multiple non-empty queues. Disciplines other than FIFO may be used for managing each individual queue.

(37) By default, one item is dequeued and made available to the receiving application through the port variable. The `Dequeue_Protocol` property specifies different dequeuing options.

- `OneItem`: (default) a single frozen item is dequeued and made available to the source text unless the queue is empty. The `Next_Value` service call has no effect.
- `AllItems`: all items that are frozen at input time are dequeued and made available to the source text via the port variable, unless the queue is empty. Individual items become accessible as port variable value through the `Next_Value` service call.
- `MultipleItems`: multiple items can be dequeued one at a time from the frozen queue and made available to the source text via the port variable. One item is dequeued and its value made available via the port variable with each `Next_Value` service call. Any items not dequeued remain in the queue and are available for the next dispatch.

(38) The `Get_Count` service call indicates how many items have been made available to the source text. A value of zero indicates that no new item is available via a data port, event port, or event port variable. A value greater than zero indicates that one or more fresh values are available.

(39) A port may have a `Fan_Out_Policy` property. This property indicates how the content is transferred through outgoing connections. The content can be passed to all recipients (`Broadcast`), or the output is distributed evenly to the recipients (`RoundRobin`), to one recipient based on content/routing information (`Selective`), or to the next recipient ready to be dispatched (`OnDemand`). `Broadcast`, `RoundRobin`, and `Selective` pass on data and events without queuing it, while `OnDemand` requires a queue that is serviced by the recipients. The size of the queue and other queue characteristics are specified as properties of the port with the fan-out.

(40) An event or event data port with a fan-out policy of `OnDemand` allows us to model a queue being serviced by multiple recipients. For example, a queue on an incoming thread group port that is connected to multiple threads allows sender output to be queued in a single queue and be serviced by multiple threads (see also Section 9.2.6).

8.3.4 Events and Subprograms

(41) Any subprogram, thread, device, or processor with an outgoing event port, i.e., **out event**, **out event data**, **in out event**, **in out event data**, can be the source of an event. During a single dispatch execution, a thread may raise zero or more events and transmit zero or more event data through `Send_Output` runtime service calls. It may also raise an event at completion through its predeclared `Complete` port (see Section 5.4) and transmit event data through event data ports that contain new values that have not been transmitted through explicit `Send_Output` runtime service calls.

(42) Events are received through **in event**, **in out event**, **in event data**, and **in out event data** ports, i.e., incoming ports. If such an incoming port is associated with a thread and the thread does not contain a mode transition naming the port, then the event or event data arriving at this port is added to the queue of the port. If the thread is aperiodic or sporadic and does not have its `Dispatch` event connected, then each event and event data arriving and queued at any incoming ports of the thread results in a separate request for thread dispatch.

Examples

```
package Patterns
thread Voter
features
  Input: in data port [3];
  Output: out data port;
end Voter;
thread Processing
features
  Input: in data port;
  Result: out data port;
end Processing;
thread group Redundant_Processing
features
  Input: in data port;
  Result: out data port;
end Redundant_Processing;
thread group implementation Redundant_Processing.basic
subcomponents
  processing: thread Processing [3];
```

```
  voting: thread voter;
connections
  port processing.Result -> voting.Input {Connection_Pattern => One_To_One;};
  port Input -> processing.Input;
  port voting.Output -> Result;
end Redundant_Processing.basic;
end Patterns;
```

8.3.5 Runtime Support For Ports

(43) The application program interface for the following services is defined in the applicable source language annex of this standard. They are callable from within the source text.

(44) A `Send_Output` runtime service allows the source text of a thread to explicitly cause events, event data, or data to be transmitted through outgoing ports to receiver ports. The `Send_Output` service takes a port list parameter that specifies for which ports the transmission is initiated. The send on all ports is considered to occur logically simultaneously. `Send_Output` is a non-blocking service. An exception is raised if the send fails with exception codes indicating the failing port and type of failure.

```
subprogram Send_Output
features
  OutputPorts: in parameter <implementation-dependent port list>;
  -- List of ports whose output is transferred
  SendException: out event data; -- exception if send fails to complete
end Send_Output;
```

NOTES: The `Send_Output` runtime service replaces the `Raise_Event` service in the original AADL standard.

(45) A `Put_Value` runtime service allows the source text of a thread to supply a data value to a port variable. This data value will be transmitted at the next `Send_Output` call in the source text or by the runtime system at completion time or deadline.

```
subprogram Put_Value
features
  Portvariable: requires data access; -- reference to port variable
  DataValue: in parameter; -- value to be stored
  DataSize: in parameter;  - size in bytes (optional)
end Put_Value;
```

(46) A `Receive_Input` runtime service allows the source text of a thread to explicitly request port input on its incoming ports to be frozen and made accessible through the port variables. Any previous content of the port variable is overwritten, i.e., any previous queue content not processed by `Next_Value` calls is discarded. The `Receive_Input` service takes a parameter that specifies for which ports the input is frozen. Newly arriving data may be queued, but does not affect the input that thread has access to (see Section 9.1). `Receive_Input` is a non-blocking service.

```
subprogram Receive_Input
    features
      InputPorts: in parameter <implementation-dependent port list>;
      -- List of ports whose input is frozen
    end Receive_Input;
```

(47) In the case of data ports the value is made available without requiring a `Next_Value` call. The `Get_Count` will return the value 1 if the value has been updated, i.e., is *fresh*. If the data is not fresh, the value zero is returned.

(48) In the case of event data ports each data value is retrieved from the queue through the `Next_Value` call and made available as port variable value. Subsequent calls to `Get_Value` or direct access of the port variable will return this value until the next `Next_Value` call.

(49) In case of event ports and event data ports the queue is available to the thread, i.e., `Get_Count` will return the size of the queue. If the queue size is greater than one the `Dequeued_Items` property and `Dequeue_Protocol` property may specify that more than one element is made accessible to the source text of a thread.

(50) A `Get_Value` runtime service shall be provided that allows the source text of a thread to access the current value of a port variable. The service call returns the data value. Repeated calls to `Get_Value` result in the same value to be returned, unless the current value is updated through a `Receive_Input` call or a `Next_Value` call.

```
subprogram Get_Value
features
  Portvariable: requires data access; -- reference to port variable
  DataValue: out parameter; -- value being retrieved
  DataSize: in parameter;  - size in bytes (optional)
end Get_Value;
```

(51) A `Get_Count` runtime service shall be provided that allows the source text of a thread to determine whether a new data value is available on a port variable, and in case of queued event and event data ports, who many elements are available to the thread in the queue. A count of zero indicates that no new data value is available.

```
subprogram Get_Count
features
  Portvariable: requires data access; -- reference to port variable
  CountValue: out parameter BaseTypes::Integer; -- content count of port variable
end Get_Count;
```

(52) A `Next_Value` runtime service shall be provided that allows the source text of a thread to get access to the next queued element of a port variable as the current value. A `NoValue` exception is raised if no more values are available.

```
subprogram Next_Value
features
  Portvariable: requires data access; -- reference to port variable
  DataValue: out parameter; -- value being retrieved
  DataSize: in parameter;  -- size in bytes (optional)
  NoValue: out event port; -- exception if no value is available
end Next_Value;
```

(53) A `Updated` runtime service shall be provided that allows the source text of a thread to determine whether input has been transmitted to a port since the last `Receive_Input` service call.

```
subprogram Updated
features
  Portvariable: in parameter <implementation-dependent port reference>;
  -- reference to port variable
  FreshFlag: out parameter BaseTypes::Boolean; -- true if new arrivals
end Updated;
```

Processing Requirements and Permissions

(54) For each data or event data port declared for a thread, a system implementation method must provide sufficient buffer space within the associated binary image to unmarshall the value of the data type. Adequate buffer space must be allocated to store a queue of the specified size for each event data port. The applicable source language annex of this standard defines data variable declarations that correspond to the data or event data features. Buffer variables may be allocated statically as part of the source text data declarations. Alternatively, buffer variables may be allocated dynamically while the process is loading or during thread initialization. A method of implementing systems may require the data declarations to appear within source files that have been specified in the source text property. In some implementations, these declarations may be automatically generated for inclusion in the final set of source text. A method of implementing systems may allow direct visibility to the buffer variables. Runtime service calls may be provided to access the buffer variables.

(55) The type mark used in the source variable declaration must match the type name of the port data component type. Language-specific annexes to this standard may specify restrictions on the form of a source variable declaration to facilitate verification of compliance with this rule.

(56) For each event or event data port declared for a thread, a method of implementing the system must provide a source name that can be used to refer to that event within source text. The applicable source language annex of this standard defines this name and defines the source constructs used to declare this name within the associated source text. A method of implementing systems may require such declarations to appear within source files that have been specified in the source text property. In some implementations, these declarations may be automatically generated for inclusion in the final set of source text.

(57) If any source text associated with a software component contains a runtime service call that operates on an event, then the enumeration value used in that service call must have a corresponding event feature declared for that component.

(58) A method of processing specifications is permitted to use non-standard property definitions and associations to define alternative queuing disciplines.

(59) A method of implementing systems is permitted to optimize the number of port variables necessary to perform the transmission of data between ports as long as the semantics of such connections are maintained. For example, the source text variable representing an out data port and the source text variable representing the connected in data port may be mapped to the same memory location provided their execution lifespan does not overlap.

Examples

```
package Nav_Types public
   data GPS properties Source_data_Size => 30 Bytes; end GPS;
   data INS properties Source_data_Size => 20 Bytes; end INS;
   data Position_ECEF properties Source_data_Size => 30 Bytes; end Position_ECEF;
   data Position_NED properties Source_data_Size => 30 Bytes; end Position_NED;
end Nav_Types;
```

```
package Navigation
process Blended_Navigation
features
   GPS_Data : in data port Nav_Types::GPS;
   INS_Data : in data port Nav_Types::INS;
   Position_ECEF : out data port Nav_Types::Position_ECEF;
   Position_NED : out data port Nav_Types::Position_NED;
properties
   -- the input rate of GPS is twice that of INS
   Input_Rate => ( 50.. 50, perSecond , Fixed ) applies to GPS_Data;
   Input_Rate => ( 100.. 100, perSecond , Fixed ) applies to INS_Data;
end Blended_Navigation;

process implementation Blended_Navigation.Simple
subcomponents
   Integrate : thread;
   Navigate : thread;
end Blended_Navigation.Simple;
end Navigation;
```

8.4 Subprogram and Subprogram Group Access

(1) Subprograms and subprogram groups can be made accessible to other components. Components can declare that they require access to subprograms and subprogram groups. Components may provide access to their subprograms and subprogram groups. Subprogram access is used to model binding of a subprogram call (local or remote) to the subprogram instance being called.

Syntax

```
-- The requires and provides subprogram access subclause
subprogram_access ::=
    defining_subprogram_access_identifier :
       ( provides |  requires ) subprogram access
        [ subprogram_unique_component_classifier_reference
        | prototype_identifier ]

subprogram_access_refinement ::=
    defining_subprogram_access_identifier : refined to
        ( provides |  requires ) subprogram access
        [ subprogram_unique_component_classifier_reference
        | prototype_identifier ]
```

```
-- The requires and provides subprogram group access subclause
subprogram_group_access ::=
    defining_subprogram_group_access_identifier :
        ( provides | requires ) subprogram group access
         [ subprogram_group_unique_component_classifier_reference
         | prototype_identifier ]

subprogram_group_access_refinement ::=
    defining_subprogram_group_access_identifier : refined to
        ( provides | requires ) subprogram group access
        [ subprogram_group_unique_component_classifier_reference
        | prototype_identifier ]
```

Naming Rules

(N1) The defining identifier of a provides or requires subprogram or subprogram group access declaration must be unique within the namespace of the component type where the subcomponent access is declared.

(N2) The defining identifier of a provides or requires subprogram or subprogram group refinement must exist as a defining identifier of a provides or requires subprogram or subprogram group or an abstract feature in the namespace of the component type being extended.

(N3) The component type identifier or component implementation name of a subprogram or subprogram group access classifier reference, if present, must exist in the package namespace.

(N4) The prototype identifier of a subprogram or subprogram group access classifier reference, if present, must exist in the namespace of the classifier that contains the access declaration.

Legality Rules

(L1) The category of the subprogram or subprogram group access declaration must be identical to the category of the component type (and of the component implementation) in the referenced subcomponent classifier or the feature being refined must be an abstract feature. In the latter case, the abstract feature must not have a direction specified.

(L2) A subprogram or subprogram group access declaration that does not specify a subcomponent classifier reference is incomplete. Such a reference can be added in a subcomponent access refinement declaration.

(L3) A subprogram or subprogram group access declaration may be refined by adding a property association. Inclusion of the subcomponent classifier reference is optional.

Consistency Rules

(C1) A *provides subprogram access* feature indicates that a subprogram is made available to be referenced. A project may enforce a consistency rule that a subprogram access connection connects this feature to directly a subprogram subcomponent, or indirectly via a *requires subprogram (group) access* or *provides subprogram (group) access*.

Standard Properties

```
-- Subprogram call rate for subprogram access
Subprogram_Call_Rate: Rate_Spec => ( Value_Range => 1.0 .. 1.0; Rate_Unit => PerDispatch;
Rate_Distribution => Fixed; )
```

```
Queue_Size: aadlinteger 0 .. Max_Queue_Size => 1

Queue_Processing_Protocol: Supported_Queue_Processing_Protocols => FIFO

Overflow_Handling_Protocol: enumeration (DropOldest, DropNewest, Error)
       => DropOldest
Urgency: aadlinteger 0 .. Max_Urgency
```

Semantics

(2) A *required subprogram (group) access* declaration indicates that a component requires access to a externally declared subprogram (group). Required subprogram (group) accesses are resolved to subprogram (group) subcomponents through access connection declarations.

(3) A *provides subprogram (group) access* declaration indicates that a component provides access to a subprogram (group) subcomponent contained in the component. Provided subprogram (group) accesses can be used to resolve required subprogram (group) accesses.

(4) A subprogram that is accessed by more than one component is shared and must be reentrant. The shared subprogram may be called by multiple threads. This may result in concurrent access to shared data components.

(5) If a different thread provides access to a subprogram then the call is remote, i.e., executed by the thread with the provides subprogram access feature. Otherwise the call is a local call, i.e., executed by the calling thread.

(6) In case of a remote subprogram call, the requesting thread calls a local proxy that carries out the service request. The proxy may marshall and unmarshall the parameters as necessary. The execution time of the client proxy is determined by the `Client_Subprogram_Execution_Time` property. The actual call results in communicating the subprogram execution request to the remote thread. While the call is in progress, the calling thread is blocked. Upon completion of the remote subprogram execution and return of results, the calling thread resumes execution by entering the ready state. A semi-synchronous remote call may be supported where the calling thread may issue the call and wait for the result at a later time by calling `Await_Result` (see Section 5.4.8). In this case the caller may issue multiple remote calls to be executed concurrently.

(7) If the called subprogram raises events or event data and the subprogram call is a remote call, then the raised event in the subprogram is mapped to the corresponding event or event data port of the caller subprogram proxy.

(8) Each provides subprogram access feature of a thread that represents an entrypoint to a remotely callable code sequence in the source text. A request for execution of such a subprogram is a dispatch request to the thread containing the subprogram. Requests for execution of subprograms may be queued if the thread is already executing a dispatch request. A thread can have multiple subprogram entrypoints, expressed by multiple subprogram access feature declarations. Only one of these subprograms may be executed per thread dispatch. Queuing and queue servicing follows the semantics of event port queues.

(9) Execution of subprogram calls may get blocked for two reasons. A call may get blocked if the call is remote to a thread that services calls and it is currently executing a dispatch, or it may get blocked because the called subprogram operates in a shared data component. This is the case, if the called subprogram is a *provides subprogram (group) access* feature of a data component that itself has shared access, i.e., is an access method of a data object, or if a shared data component is accessible to the subprogram through a requires data access feature of the subprogram. In the former case the thread servicing the calls assures mutual exclusion, while other remote calls to subprograms of the thread are queued. In the latter case, concurrent access to the data component is assured to be mutually exclusive according to the `Concurrency_Control_Protocol` property value and realized through the `Get_Resource` service call in the source text, while other mutually exclusive access attempts to shared data components are queued.

(10) `Call_Rate` specifies the number of times per dispatch or per second at which a subprogram is called.

Examples

```
-- a remote procedure call from one thread to another thread
package RemoteCallExample
public
system implementation simple.impl
subcomponents
   A: process caller_P.i;
   B: process remote_P.i;
connections
   subprogram access A.DoCalc -> B.DoCalc;
end simple.impl;

process remote_P
features
   DoCalc: provides subprogram access Calc;
end remote_P;

process implementation remote_P.i
subcomponents
   t1: thread Remote;
   -- other subcomponent declarations
connections
   subprogram access t1.MyCalc -> DoCalc;
end remote_P.i;

thread Remote
features
   MyCalc: provides subprogram access Calc;
end Remote;

process caller_P
features
   DoCalc: requires subprogam access Calc;
end caller_P;

process implementation caller_P.i
subcomponents
   Q: thread caller;
connections
```

```
   subprogram access DoCalc -> Q.MyCalc;
end caller_P.i;

thread caller
features
   MyCalc: requires subprogram access Calc;
end caller;
end RemoteCallExample;

-- A Printer Server Example
package PrinterServerExample
public
   process printers
   features
      printonServer: provides subprogram access print;
      mainPrinter: in event port;
      backupPrinter: in event port;
   end printers;

   process implementation printers.threaded
   subcomponents
      A : thread printer in modes ( modeA );
      B : thread printer in modes ( modeB );
  connections
      printtoA: subprogram access A.print -> printonServer in modes (modeA);
      printtoB: subprogram access B.print -> printonServer in modes (modeB);
  modes
      modeA: initial mode;
      modeB: mode;
     modeA -[ backupPrinter ]-> modeB;
     modeB -[ mainPrinter ]-> modeA;
   end printers.threaded;

   thread printer
   features
      print : provides subprogram access print;
   end printer;

   subprogram print
```

```
   features
      filetoprint: in parameter file;
   end print;

   data file
   end file;

  process ApplicationSystem
  end ApplicationSystem;

  process implementation ApplicationSystem.default
  subcomponents
   app: process ApplicationSystem;
   printserver: process PrintServer;
  connections
   subprogram access printserver.print -> app.print;
  end ApplicationSystem.default;
end PrinterServerExample;
```

8.5 Subprogram Parameters

(1) Subprogram parameter declarations represent data values that can be passed into and out of subprograms. Parameters are typed with a data classifier reference representing the data type.

Syntax

```
parameter ::=
    defining_parameter_identifier :
        ( in | out | in out ) parameter
        [ data_unique_component_classifier_reference |
          prototype_identifier ]

parameter_refinement ::=
    defining_parameter_identifier : refined to
        ( in | out | in out ) parameter
        [ data_unique_component_classifier_reference |
          prototype_identifier ]
```

Naming Rules

(N1) The defining identifier of a parameter must be unique within the namespace of the subprogram type containing the parameter declaration.

(N2) The defining parameter identifier of a parameter refinement declaration must also appear in a feature declaration of a component type being extended and must refer to a parameter or an abstract feature.

(N3) The data classifier reference must refer to a data component type or a data component implementation.

(N4) The prototype identifier, if present, must exist in the namespace of the subprogram classifier that contains the parameter declaration.

Legality Rules

(L1) Parameters can be declared for subprogram component types.

(L2) A parameter declaration that does not specify a data classifier reference is incomplete. Such a reference can be added in a parameter refinement declaration.

(L3) A parameter declaration may be refined by adding a property association. Inclusion of the data classifier reference is optional.

(L4) The parameter direction of a parameter refinement must be the same as the direction of the feature being refined. If the feature being refined is an abstract feature without direction, then all parameter directions are acceptable.

Standard Properties

```
-- Properties specifying the source text representation of the parameter
Source_Name: aadlstring
Source_Text: inherit list of aadlstring
```

Semantics

(2) A subprogram parameter specifies the data that are passed into and out of a subprogram. The data type specified for the parameter and the data type of the actual data passed to a subprogram must be compatible according to the `Classifier_Matching_Rule`.

(3) An **in out** parameter declaration represents a parameter whose value is passed in and returned by value. Parameters passed by reference are modeled using **requires data access**.

8.6 Data Component Access

(1) Data component access is used to model shared data. Data components can be made accessible outside their containment hierarchy. Components can declare that they require access to externally declared data components. Components may provide access to their data components.

(2) The use of component access for data components is illustrated in Figure 12. Data2, Thread1, and Thread2 are subcomponents of a process implementation. Thread1 contains a data subcomponent called Data1. Data1 is made accessible outside Thread1 through a **provides data access** feature declaration in the thread type of Thread1. It is being accessed by Thread2 as expressed by a **requires data access** feature declaration in the thread type of Thread2. Thread1 accesses data component Data2.

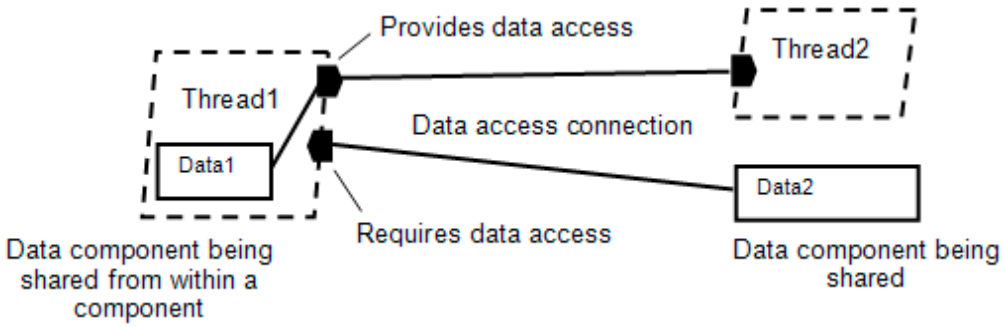


Figure 12 Containment Hierarchy and Shared Access

Syntax

```
-- The requires and provides subcomponent access subclause
data_access ::=
    defining_data_component_access_identifier :
        ( provides |  requires ) data access
        [ data_unique_component_classifier_reference
         | prototype_identifier ]

data_access_refinement ::=
    defining_data_component_access_identifier : refined to
        ( provides |  requires ) data access
        [ data_unique_component_classifier_reference
        | prototype_identifier ]
```

Naming Rules

(N1) The defining identifier of a provides or requires data access declaration must be unique within the namespace of the component type where the data access is declared.

(N2) The defining identifier of a provides or requires data access refinement must exist as a defining identifier of a provides or requires data access or as a defining identifier of an abstract feature in the namespace of the component type being extended.

(N3) The component type identifier or component implementation name of a data access classifier reference must exist in the package namespace.

(N4) The prototype identifier, if present, must exist in the namespace of the classifier that contains the data access declaration.

Legality Rules

(L1) The data access classifier reference must be of category **data**.

(L2) A data access declaration may be refined by refining the data classifier, by adding a property association, or by doing both. If the refinement only adds a property association the subcomponent classifier reference is optional.

Consistency Rules

(C1) A data access declaration that does not specify a data classifier reference is incomplete. Such a reference can be added in a data access refinement declaration.

(C2) If the source code of a component does access shared data, then the component type declaration must specify a requires data access declaration. In other words, for all components that access shared data their component type declaration must reflect that fact.

(C3) A data access refinement may refine an abstract feature declaration. If the abstract feature declaration specifies a direction of **in**, then the access right of the data access must be read-only. If the direction is **out**, then the access right of the data access must be write-only. If the abstract feature does not have a specified direction, then any access right is acceptable.

Standard Properties

```
Access_Right : Access_Rights => read_write
-- access time range for data access
Access_Time: record (
    First:  IO_Time_Spec ;
    Last: IO_Time_Spec ; )
    => ( First =>(Time => Start; Offset => 0.0 ns .. 0.0 ns;);
      Last => (Time => Completion; Offset => 0.0 ns .. 0.0 ns;); )
```

Semantics

(3) A *requires data access* declaration in the component type indicates that a component requires access to a component declared external to the component. Required data accesses are resolved to actual data subcomponents through access connection declarations. For data components different forms of required access, such as read-only access, are specified by a `Access_Right` property. Read-only access can also be specified through a directional access connection from the data component to the *requires data access* feature, while write-only access is specified through a directional access connection to the data component.

(4) A *provides data access* declaration in the component type indicates that a subcomponent provides access to a data component contained in the component. Provided data accesses can be used to resolve required subcomponent access. For data and bus components different forms of provided access, such as read-only access, are specified by a `Access_Right` property or be directional access connections.

(5) If a data access feature is a refinement of an abstract feature, then the direction of the abstract feature, if specified, imposes a restriction on the data flow, i.e., **in** implies read-only, and **out** implies write-only.

(6) Shared data may be accessed by multiple threads. Such potential concurrent access is controlled according to the `Concurrency_Control_Protocol` property.

(7) `Access_Time` specifies the time range over which a component has access to a shared data component. By default access is required for the duration of the component execution. The value of a shared data component is read or written through the use of a data variable that represents the shared data component, or through `Get_Value` and `Put_Value` service calls. Write access immediately updates the shared data component.

Examples

```
package Example
system simple
end simple;

system implementation simple.impl
subcomponents
   A: process pp.i;
   B: process qq.i;
connections
   data access A.dataset -> B.Reqdataset;
end simple.impl;
```

```
process pp
features
   Dataset: provides data access dataset_type;
end pp;

process implementation pp.i
subcomponents
   Share1: data dataset_type;
   -- other subcomponent declarations
connections
   data access Share1 <-> Dataset;
end pp.i;

process qq
features
   Reqdataset: requires data access dataset_type;
end qq;

process implementation qq.i
subcomponents
   Q: thread rr;
connections
   data access Reqdataset <-> Q.Req1;
end qq.i;

thread rr
features
   Req1: requires data access dataset_type;
end rr;
end Example;
```


8.7 Bus Component Access

(1) Bus components can be made accessible to other components. Components can declare that they require access to externally declared buses. Components may provide access to their buses. Bus access is used to model connectivity of execution platform components through buses.

(2) Figure 13 illustrates the use of a shared bus. Bus1 provides the connection between Processor1, Memory1, and Device1. In addition, the bus is being made accessible outside System1.

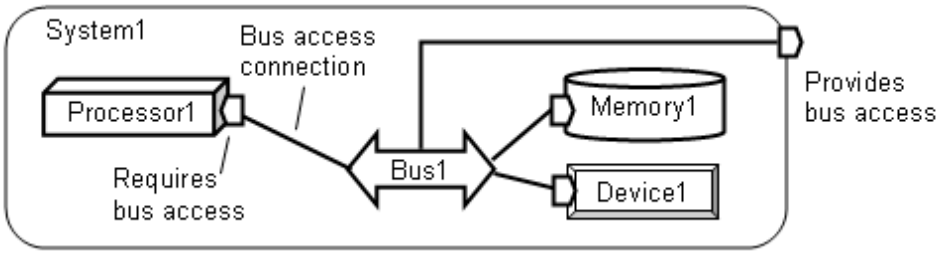


Figure 13 Shared Bus Access

Syntax

```
-- The requires and provides bus access subclause
bus_access ::=
    defining_bus_access_identifier :
        ( provides |  requires ) bus access
         [ bus_unique_component_classifier_reference
         | prototype_identifier ]

bus_access_refinement ::=
    defining_bus_access_identifier : refined to
        ( provides |  requires ) bus access
        [ bus_unique_component_classifier_reference
        | prototype_identifier ]
```

Naming Rules

(N1) The defining identifier of a provides or requires bus access declaration must be unique within the namespace of the component type where the bus access is declared.

(N2) The defining identifier of a provides or requires bus refinement must exist as a defining identifier of a requires or provides bus access or of an abstract feature in the namespace of the component type being extended.

(N3) The component type identifier or component implementation name of a bus access classifier reference must exist in the package namespace.

(N4) The prototype identifier, if present, must exist in the namespace of the classifier that contains the bus access declaration.

Legality Rules

(L1) The data access classifier reference must be of category **bus**.

(L2) A bus access declaration may be refined by refining the bus classifier, by adding a property association, or by doing both. If the refinement only adds a property association the bus classifier reference is optional.

Consistency Rules

(C1) A bus access declaration that does not specify a bus classifier reference is incomplete. Such a reference can be added in a bus access refinement declaration.

(C2) If a bus access feature is a refinement of an abstract feature, then the direction of the abstract feature, if specified, imposes a restriction on the access right, i.e., **in** implies read-only, and **out** implies write-only.

Standard Properties

```
Access_Right : Access_Rights => read_write
```

Semantics

(3) A *required bus component access* declaration in the component type indicates that a component requires access to a component declared external to the component. Required bus accesses are resolved to actual bus subcomponents through access connection declarations.

(4) A *provides bus access* declaration in the component type indicates that a subcomponent provides access to a bus contained in the component. Provided bus accesses can be used to resolve required bus access. For bus components different forms of provided access, such as read-only access, are specified by a `Access_Right` property or by directional access connections.

(5) A bus that is accessed by more than one component is shared. The shared bus is a common resource through which processor, memory, and device components communicate.

Examples

```
package Example2
system simple
end simple;

system implementation simple.impl
subcomponents
   A: processor Hardware::PPC;
   B: device Equipment::DigCamera;
connections
   bus access A.USB1 <-> B.USB2;
end simple.impl;

processor PPC
features
   USB1: provides bus access Hardware::USB;
end PPC;

device DigCamera
properties
   USB2: requires bus access USB;
end DigCamera;
end Example2;
```

9 Connections

(1) A *connection* is a linkage between features of two components that represents communication of data and control between components. This can be the transmission of control and data between ports of different threads or between threads and processor or device components. A connection may denote an event that triggers a mode transition. The timing of data and control transmission depends on the connection category and on the dispatch protocol of the connected threads. The flow of data between parameters of subprogram calls within a thread may be specified using connections. Finally, connections designate access to shared components.

(1) The AADL supports connections between abstract features, feature groups connections, port connections, parameter connections, and access connections. Port connections represent directional flow of data and control between two concurrently executing components, i.e., between two threads or between a thread and a processor or device. Parameter connections denote the flow of data through the parameters of a sequence of subprogram calls within a thread. Data access connections designate access to shared data components by concurrently executing threads or by subprograms executing within a thread. Bus access connections represent communication between processors, memory, and devices by accessing a shared bus. Subprogram access connections represent the binding of a subprogram call to a subprogram instance being called.

(2) When an AADL model with a thread architecture is instantiated, a connection instance is created from the ultimate source to the ultimate destination component by following a sequence of connection declarations. The ultimate source and ultimate destination typically are the active components in the instance model or components whose access is shared. This connection instance is referred to as semantic connection and its semantic details are defined for each of the different types of connections. If the AADL model is not fully detailed to the thread level, connection instances are created between the leaf components in the system hierarchy.

Syntax

```
connection ::=
    [ defining_connection_identifier : ]

    ( feature_connection
    | port_connection
    | parameter_connection
    | access_connection
    | feature_group_connection )
    [ { { property_association }+ } ]
    [ in_modes_and_transitions ] ;


connection_refinement ::=
    defining_connection_identifier : refined to

    [ feature_connection_refinement
    | port_connection_refinement
    | parameter_connection_refinement
    | access_connection_refinement
    | feature_group_connection_refinement ]
    [ { { property_association }+ } ]
    [ in_modes_and_transitions ] ;
```

Naming Rules

(N1) The defining identifier of a defined connection declaration must be unique in the local namespace of the component implementation with the connection subclause.

(N2) The connection identifier in a connection refinement declaration must refer to a named connection declared in an ancestor component implementation.

(N3) For mode-specific connections, as indicated by the `in_modes` subclause, a connection name may appear more than once.

Legality Rules

(L1) A connection refinement must contain at least one of the following: a connection source and destination subclause, a property association, an **in modes** subclause.

(L2) If a semantic connection may be active in a particular mode, then the ultimate source and ultimate destination components must be part of that mode.

(L3) If a semantic connection may be active in a particular mode transition, then the ultimate source component must be part of a system mode that includes the old mode identifier and the ultimate destination component must be part of a system mode that includes the new mode identifier.

Semantics

(3) Connections define directional and bidirectional connectivity between abstract features, ports, access features, parameters, and between feature groups.

(4) Connections can have properties. Connections can be defined to be active in certain modes or in mode transitions, as indicated by the **in modes** subclause.

9.1 Feature Connections

(1) A feature connection can be declared between two abstract features of components. If the features specify a direction, then the source of the connection is the feature with the **out** direction and the destination is the feature with the **in** direction. Otherwise, the connection is considered to be bidirectional.

(2) A feature connection can also be declared to refer to concrete features, i.e., ports, parameters, and access features. In this case, the type of connection is inferred from the categories of the features involved in the connection.

Syntax

```
feature_connection ::=
    source_feature_reference connection_symbol
    destination_feature_reference

connection_symbol ::=
    directional_connection_symbol | bidirectional_connection_symbol

directional_connection_symbol ::= ->

bidirectional_connection_symbol ::= <->
```

```
feature_reference ::=
    -- feature in the component type
    component_type_feature_identifier |
    -- feature in a feature group of the component type
    component_type_feature_group_identifier . feature_identifier |
    -- feature in a subcomponent
    subcomponent_identifier . feature_identifier


feature_connection_refinement ::=
    source_feature_reference connection_symbol
    destination_feature_reference
```

Naming Rules

(N1) A source or destination reference in a feature connection or feature connection refinement declaration must reference a feature declared in the component type, a feature in a feature group of the component type, or a feature of one of the subcomponents. The source and destination features must be abstract features, ports, parameters, or access features.

(N2) The subcomponent reference may refer to a subcomponent or a subcomponent array.

Legality Rules

(L1) The feature identifier of a subcomponent reference may refer to a feature array, if the subcomponent is a thread, device, or processor.

The direction declared for the destination feature of a feature connection declaration must be compatible with the direction declared for the source feature as defined by the following rules:

(L2) If the feature connection declaration represents a connection between features of sibling components, then the source must be an outgoing feature and the destination must be an incoming feature. An *outgoing feature* is an abstract feature with no direction or **out** direction, an outgoing port or parameter with an **out** or **in out** direction, or an access feature, in the case of data access with at least write access. An *incoming feature* is an abstract feature with no direction or **in** direction, an incoming port or parameter with an **in** or **in out** direction, or an access feature, in the case of data access with at least read access.

(L3) If the feature connection declaration represents a connection between features up the containment hierarchy, then the source and destination must both be outgoing features.

(L4) If the feature connection declaration represents a connection between features down the containment hierarchy, then the source and destination must both be incoming features.

(L5) If the feature connection declaration specifies a directional connection, then the direction of the connection must be supported by the direction of the source and destination features.

(L6) The individual connections of a semantic connection must be bidirectional or have the same direction. The direction of the connection is determined by the direction of the source and destination feature and by the direction of the connection declarations.

Standard Properties

```
Required_Virtual_Bus_Class : inherit list of classifier (virtual bus)
Required_Connection_Quality_Of_Service : inherit list of Supported_Connection_QoS
Connection_Pattern: list of Supported_Connection_Patterns
Connection_Set: list of Connection_Pair
```

Semantics

(3) Feature connections represent connections between abstract features, or between an abstract feature and a concrete feature. Connection patterns are specified as described in Section 9.2.3.

9.2 Port Connections

(1) Port connections represent directional transfer of data and control between two concurrently executing components, i.e., between threads, processors, and devices. They are feature connections, whose source and destination are limited to ports and data access.

(2) These connections are *semantic port connections*. A semantic port connection is determined by a sequence of one or more individual port connection declarations that follow the component containment hierarchy in a fully instantiated system from an *ultimate source* to an *ultimate destination*. In a partial AADL model the ultimate source and destination of a semantic port connection are the ports of leaf components in the containment hierarchy, i.e., a thread group, process, or system without subcomponent.

(3) Semantic port connections are illustrated in Figure 14. The *ultimate source* of a semantic port connection is an outgoing port of a thread, virtual processor, processor, or device component, or is a data component. The *ultimate destination* of a semantic port connection is an incoming port of a thread, virtual processor, processor, or device component, or is a data component. In the case of a bidirectional connection between **in out** ports either port can be the ultimate source or destination.

(4) Port connection declarations follow the containment hierarchy of threads, thread groups, processes and systems, or devices, processors, and systems. An individual port connection declaration links an outgoing port of one subcomponent to an incoming port of another subcomponent, i.e., it connects sibling components at the highest level in the component hierarchy required for the connection. Alternatively, a port connection declaration maps an outgoing port of a subcomponent to an outgoing port of a containing component or an incoming port of a containing component to an incoming port of a subcomponent. In other words, these connections traverse up and down the containment hierarchy.

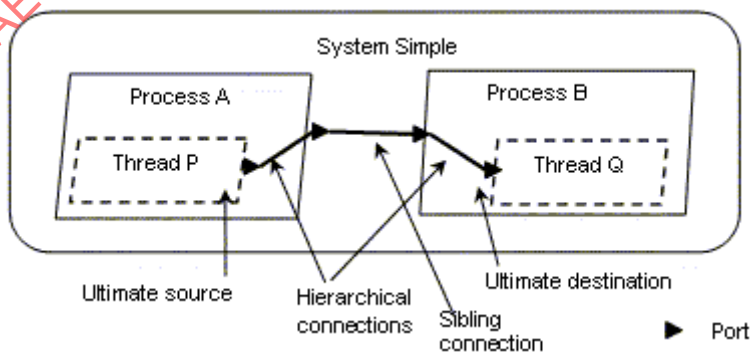


Figure 14 Semantic Port Connection

(5) Semantic port connections also represent the sampling of a data component content by a data or event data port, and updating a data component with the output of a data or event data port. In other words, the ultimate source or the ultimate destination of a semantic port connection, but not both, can be a data component.

(6) Semantic port connections may also route a raised event to a modal component through a sequence of connection declarations. A mode transition in such a component is the ultimate destination of the connection, if the mode transition names an incoming event port in the enclosing component, or an outgoing event port of one of the subcomponents (see Section 12).

(7) Semantic port connections may exist between arrays of component instances. In this case, the `Connection_Pattern` or `Connection_Set` property specifies which source array elements have a semantic connection to which destination array element (see section 9.2.3 for more detail).

(8) This section defines the concepts of departure and arrival times of port connection transmission for each type of *port connection*. The transfer semantics between periodic threads can be defined such that they ensure deterministic sampling of data. All other communication can be defined to be sampling or data driven. The inputs and outputs can be specified to occur at dispatch, any time during execution, at completion, or at deadline.

Syntax

```
port_connection ::=
    port
      source_port_connection_reference
          connection_symbol
       destination_port_connection_reference



port_connection_refinement ::=
    port
       [ source_port_connection_reference connection_symbol
         destination_port_connection_reference ]

port_connection_reference ::=
        -- port in the component type
      component_type_port_identifier
    |
        -- port in a subcomponent
      subcomponent_identifier . port_identifier
    |
        -- port element in a feature group of the component type
      component_type_feature_group_identifier . element_port_identifier
    |
        -- data element in aggregate data port
      component_type_port_identifier . data_element_identifier
    |
        -- requires data access in the component type
      component_type_requires_data_access_identifier
    |
        -- data subcomponent
```

```
    data_subcomponent_identifier
  |
    -- data component provided by a subcomponent
    subcomponent_identifier . provides_data_access_identifier
  |
    -- data access element in a feature group of the component type
   component_type_feature_group_identifier . element_data_access_identifier
  |
    -- access to element in a data subcomponent
    data_subcomponent_identifier .data_subcomponent_identifier
  |
    -- processor port
    processor . processor_port_identifier
  |
    -- component itself as event or event data source
    self . event_or_event_data_source_identifier
-- Note: data port, event data port, and event port connections
-- are replaced by port connections in AADL V2
```

Naming Rules

(N1) The connection identifier in a port connection refinement declaration must refer to a named port or feature connection declared in an ancestor component implementation.

(N2) A source or destination reference in a port connection or port connection refinement declaration must reference a port declared in the component type, a port of one of the subcomponents, or a port that is an element of a feature group in the component type, or it must refer to a requires data access in the component type, a provides data access in one of the subcomponents, or a data subcomponent.

(N3) The subcomponent reference may also consist of a reference to a subcomponent array.

(N4) The event identifier of an event source specification (**self**.*event*_identifier) must be unique among event source or destination identifiers of the given component.

Legality Rules

(L1) The ultimate source of a directional semantic port connection must be an outgoing port of a thread, virtual processor, processor, or device, or must be a data component. The ultimate destination of a directional semantic port connection must be an incoming port of a thread, virtual processor, processor, or device, or must be a data component, or a mode transition of a system, process, thread group, thread, virtual processor, processor, or device. In the case of a bidirectional port connection both ports must be **in out** ports or a data component with `read_write` access. The source feature referenced in a port connection declaration must be a feature of a thread, thread group, process, virtual processor, processor, device, or system component or a data component. The destination feature referenced in a port connection declaration must be a feature of a thread, thread group, process, virtual processor, processor, device, or system component.

(L2) The feature identifier of a subcomponent reference may refer to a feature array, if the subcomponent is a thread, device, or processor.

(L3) The following are acceptable sources and destinations of port connections. The left column shows connections between ports, while the right column shows connection between ports and data components.

event port -> event port data port -> data port, event data port, event port event data port -> event data port, data port, event port	data, data access -> data port, event data port, event port data port, event data port -> data, data access

The direction declared for the destination port of a port connection declaration must be compatible with the direction declared for the source port(s) as defined by the following rules:

(L4) If the port connection declaration represents a connection between ports of sibling components, then the source must be an outgoing port and the destination must be incoming ports.

(L5) If the port connection declaration represents a connection between ports up the containment hierarchy, then the source and destination must both be outgoing ports.

(L6) If the port connection declaration represents a connection between ports down the containment hierarchy, then the source and destination must both be incoming ports.

(L7) If the port connection declaration represents a connection between a data component or data access feature and a port, then the data component must have the following access right: as source the access right must be read-only or read-write; as destination the access right must be write-only or read-write.

(L8) The individual connections of a semantic port connection must be bidirectional or have the same direction. The direction of the connection is determined by the direction of the source and destination feature and by the direction of the connection declarations.

(L9) **Self.**<identifier> is outgoing must only be referenced as the source of a connection.

(L10) A data port cannot be the destination of more than one semantic port connection unless each semantic port connection is contained in a different mode.

(L11) A semantic connection cannot contain connection declarations with both immediate and delayed `Timing` property values.

(L12) If a port connection refinement specifies a source and a destination, then the source and destination must be identical to those of the connection being refined, or the source or destination being refined must be an abstract feature.

(L13) A port connection refinement may refine the connection direction from bidirectional to directional.

(L14) For connections between data ports, event data ports and data access, the data classifier of the source port must match the data type of the destination port. The `Classifier_Matching_Rule` property specifies the rule to be applied to match the data classifier of a connection source to the data classifier of a connection destination.

(L15) The following rules are supported:

- `Classifier_Match`: The source data type and data implementation must be identical to the data type or data implementation of the destination. If the destination classifier is a component type, then any implementation of the source matches. This is the default rule.

- `Equivalence`: An indication that the two classifiers of a connection are considered to match if they are listed in the `Supported_Classifier_Equivalence_Matches` property. Acceptable data classifier matches are specified as `Supported_Classifier_Equivalence_Matches` property with pairs of classifier values representing acceptable matches. Either element of the pair can be the source or destination classifier. Equivalence is intended to be used when the data types are considered to be identical, i.e., no conversion is necessary. The `Supported_Classifier_Equivalence_Matches` property is declared globally as a property constant.
- `Subset`: A mapping of (a subset of) data elements of the source port data type to all data elements of the destination port data type. Acceptable data classifier matches are specified as `Supported_Classifier_Subset_Matches` property with pairs of classifier values representing acceptable matches. The first element of each pair specifies the acceptable source classifier, while the second element specifies the acceptable destination classifier. The `Supported_Classifier_Subset_Matches` property is declared globally as a property constant. A virtual bus or bus must represent a protocol that supports subsetting, such as OMG DDS.
- `Conversion`: A mapping of the source port data type to the destination port data type, where the source port data type requires a conversion to the destination port data type. Acceptable data classifier matches are specified as `Supported_Type_Conversions` property with pairs of classifier values representing acceptable matches. The first element of each pair specifies the acceptable source classifier, while the second element specifies the acceptable destination classifier. The `Supported_Type_Conversions` property may be declared globally as a property constant. A virtual bus or bus must support the conversion from the source data classifier to the destination classifier.

(L16) If more than one port connection declaration in a semantic port connection has a property association for a given connection property, then the resulting property values must be identical.

(L17) A processor port specification must only be used in event connections within threads and subprograms.

Consistency Rules

(C1) There cannot be cycles of immediate connections between threads, devices, and processors.

(C2) The processor port identifier of a processor port specification (**processor**.*processor_port*_identifier) must name a port of the processor that the thread is bound to.

(C3) The `Supports_Classifier_Subset_Matches` property may be associated with a bus or virtual bus. This specifies the subset matches a particular protocol supports. Subset matches of connections bound to such a virtual bus or bus must be supported by the respective virtual bus or bus.

(C4) The `Supports_Type_Conversions` property may be associated with a bus or virtual bus. This specifies the subset matches a particular protocol supports. Subset matches of connections bound to such a virtual bus or bus must be supported by the respective virtual bus or bus.

Standard Properties

```
Timing : enumeration (sampled, immediate, delayed) => sampled

Connection_Pattern: list of Supported_Connection_Patterns

Connection_Set: list of Connection_Pair

Transmission_Type: enumeration ( push, pull )

Required_Connection_Quality_Of_Service : inherit list of Supported_Connection_QoS

Allowed_Connection_Binding_Class:
   inherit list of classifier(processor, virtual processor, bus, virtual bus, device,
memory)

Allowed_Connection_Binding: inherit list of reference (processor, virtual processor, bus,
virtual bus, device, memory)

Not_Collocated: record (
```

```
    Targets: list of reference (data, thread, process, system, connection);
    Location: classifier ( processor, memory, bus, system ); )
Actual_Connection_Binding: inherit list of reference (processor, virtual processor, bus,
virtual bus, device, memory)
```

Semantics

9.2.1 Port Connection Characteristics

(9) A semantic port connection represents directed flow of data and control between threads, processors and devices. In the case of event or event data ports the ultimate destination can be a mode transition of a component.

(10) A port connection can refine a feature connection. A port connection can be refined by changing the direction from bidirectional to directional, by adding a **in modes** subclause, and by adding property associations for the connection in a connection refinement declaration.

(11) A port connection declared with the optional `in_modes_and_transitions` subclause specifies whether the connection is part of specific modes or is part of the transition between two specific modes. The detailed semantics of this subclause are defined in Section 12.

(12) While in a given mode, transmission over a port connection only occurs if the connection is part of the current mode.

(13) During a mode switch, transmission over a data port connection only occurs at the actual time of mode switch if the port connection is declared to apply to the transition between two specific modes. The actual mode switch initiates transmission. This allows data state to be transferred between threads active in different modes. Similarly, for event or event data ports it allows for transfer of queue content.

(14) Port connections may refer to an event source specification (**self**.eventname). An event source specification indicates that the component itself is the source of an event. In the case of a thread this may be due to a *Send_Output* system call or due to an event raised by the underlying runtime system, i.e., the processor. In the case of thread groups, processes, and systems it may represent the fact that a component fault is the source of an event, as specified by the Error Model Annex (see Annex Document C).

(15) A thread or device may be connected to the port of a processor. For example, a health monitoring thread may receive the heart beat events from several processors. In addition, a port connection may refer to the port of the processor that an application software component is bound to (**processor** . <portname>). If a processor or device is the data connection source, then the transmission is initiated and completed when the destination thread or device is dispatched. In this case a data port can model a processor or device register that is sampled by a thread or device. If a device or processor is the event connection source, then the occurrence of an interrupt represents the initiation of an event transmission.

(16) AADL supports the following port connection declarations:

- Event port, data port, event data port, data (data access) -> event port: port output or written data is recognized as event and queued in the event port.
- Event data port, data port, data (data access) -> event data port: data output or written data is transferred and received as event data in a queued port.
- Data port, event data port, data (data access) -> data port: Data output or written data is transferred and available upon receipt as most recent value of a data port variable, i.e., the data port samples data.

(17) A port connection to a mode transition is declared by naming the event port or event data port in the mode transition declaration (see Section 12).

9.2.2 Port Connection Topology

(18) The AADL supports n-to-n connectivity for event and event data ports. A port may have multiple outgoing connections, i.e., its content is transmitted to multiple destinations. This means that each destination port receives an instance of the event, or event data being transmitted. Similarly, event and event data ports can support multiple incoming connections resulting in sequencing and possibly queuing of incoming events and event data.

(19) For example, multiple threads may connect their outgoing event data port to an enclosing process event data port (fan-in), this port is connected to an incoming process event data port, and this port is connected to multiple thread ports (fan-out). This results in semantic connections from all ultimate source threads to all ultimate destination threads. If the port connections from multiple threads are declared to a feature group in the enclosing process, a feature group connection to a second process, and port connections from the feature group of the second process to its contained threads, the result is a collection of pair wise semantic connections from the ultimate source threads to the ultimate destination threads.

(20) Data ports are restricted to 1-n connectivity, i.e., a data port can have multiple outgoing connections, but only one incoming connection per mode. Since data ports hold a single data state value, multiple incoming connections would result in multiple sources overwriting each other's values in the destination port variable.

(21) Data ports can be used to model aggregate data ports. If data ports are declared with a data component type that has its data subcomponents externally visible through provides data access declarations, then a separate connection can be declared to each of these elements in the port of the enclosing component. For example, several periodic threads can deliver their data port results to a data port of the enclosing process that represents the aggregate of those data values as elements of its data component type. The set of threads whose output is considered are those whose dispatch aligns. Once they have produced their output, the aggregate output is sent to the recipients. The time at which the send is initiated is the latest completion of the source threads, i.e., max($t_{complete}$), where $t_{complete}$ represents the value of t at completion time. Similarly, the recipient thread may receive an element of the aggregate data port of its enclosing process, or a subset of the elements. The latter is modeled by the classifier of the recipient data port satisfying the `Subtype` or `Subset` matching rules of the `Classifier_Matching_Rule` property.

(22) Feature groups may have multiple outgoing and incoming connections unless any ports that are elements of a feature group place additional restrictions. A destination feature group may be a subset of the source feature group. Acceptable matches are specified via the `Classifier_Matching_Rule` property with values `Subtype` or `Subset`.

(23) If a component has an **in out** port, this port may be the destination of a connection from one component and the source of a connection to another component. This can be expressed by two directional port connections. Bidirectional flow between two components is represented by a bidirectional port connection between the **in out** ports of two components.

9.2.3 Connection Patterns for Component Arrays and Feature Arrays

(24) The `Connection_Pattern` property specifies how semantic connections are replicated if the ultimate source and ultimate destination are component arrays. The ultimate source or destination is a component array if it or any enclosing component involved in the connection is declared as subcomponent array. The dimensions of the ultimate source and destination array is the sum of dimensions of the ultimate source/destination component and those of any enclosing component involved in the semantic connection. The order of the dimensions is from the ultimate source/destination component up the containment hierarchy.

(25) The ultimate source or ultimate destination of a semantic connection may be a feature array. In this case, the dimension of the feature array is treated as an additional dimension (the first dimension if multiple dimensions exist).

(26) The `Connection_Pattern` property is a multi-valued list of dimension pattern values. A dimension pattern value is a list itself with one value for each of the dimensions of component arrays. The number of dimension pattern values must correspond to the larger dimensionality of the source or destination component array. Each value specifies the intended connectivity for one dimension of the array. The following connection patterns are predefined for an array dimension:

- An `All_To_All` value indicates that each array element of the ultimate source has a semantic connection to each element in the ultimate destination (broadcast). This connection pattern applies even if the two arrays have different sizes.
- A `One_To_One` value indicates that elements of the ultimate source array and the ultimate destination array have pair wise semantic connections (Identity). This property value applies if the two arrays have identical sizes. If one range is a subset of the other then only the subset starting with the first element is connected.
- A `Next` or `Previous` value indicates that elements of the ultimate source array dimension are connected to the next (previous) element in the ultimate destination array dimension. The last element does not connect in the case of next and the first element does not connect in the case of previous. This property value applies if the two arrays have identical sizes. Note that a `Next` value for two dimensions results in a diagonal connection.
- A `Cyclic_Next` or `Cyclic_Previous` value indicates that elements of the ultimate source array dimension are connected to the next (previous) element in the ultimate destination array dimension. In the case of `Cyclic_Next` the last element in the array connects to the first, and vice versa for `Cyclic_Previous`. This property value applies if the two arrays have identical sizes. Note that a `Next` value for two dimensions results in a diagonal connection.
- A `One_to_All` value indicates that a single element of the ultimate source has a semantic connection to each element in the ultimate destination. This connection pattern is used when the destination array has a higher dimensionality than the source array. It specifies that any connection according to the other dimensions is being replicated for each element in this destination dimension, i.e., the outputs are broadcast in this dimension.
- An `All_to_One` value indicates that each array element of the ultimate source has a semantic connection to a single element in the ultimate destination. This connection pattern is used when the destination array has a lower dimensionality than the source array. It specifies that any connection according to the other dimensions is being replicated for each element in this source dimension, i.e., the outputs of this dimension are connected to a single fan-in point.

(27) A list of `Connection_Pattern` values permits more complex patterns to be defined. Figure 15 illustrates the use of connection patterns in a two-dimensional array. In more complex patterns, the value of `((Next,One_To_One), (One_To_One,Next), (Previous, One_To_One), (One_To_One,Previous))` indicates connections to all horizontal and vertical neighbors, while `((Next,One_To_One), (One_To_One,Next), (Previous, One_To_One), (One_To_One,Previous), (Previous,Previous), (Next,Previous), (Previous,Next), (Next,Next))` includes diagonal neighbors as well.

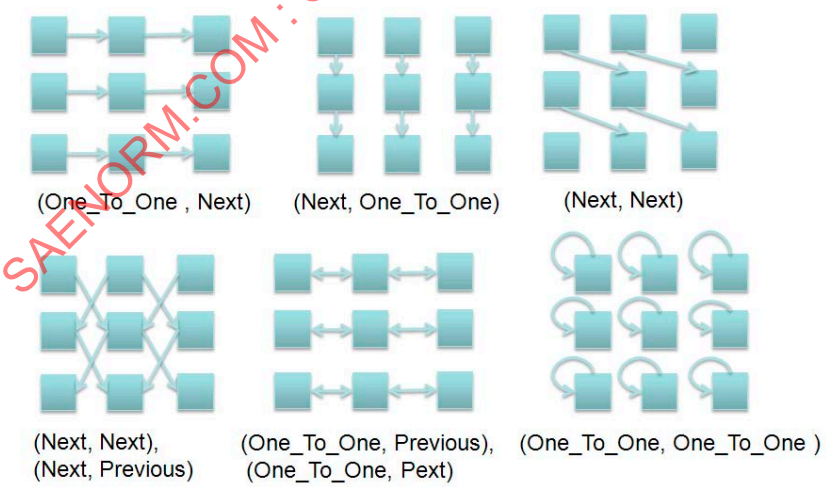


Figure 15 Connection Patterns in 2-Dimensional Component Array

(28) The `Connection_Set` property specifies each semantic connection between elements of the source component array and destination component array individually. The property has a list of pairs of source and destination array indices. A source or destination array index consists of a list **aadlinteger** values, one for each array dimension. The first index is the value 1. The values for the `Connection_Set` property may be generated by a tool based on pattern specification that is an annex extension of the core AADL.

(29) If both `Connection_Pattern` values and `Connection_Set` values are specified, the set of semantic connections is the union of both.

Examples

```
-- The first sensor array in Figure 15
device sensor
features
  Incoming: in event data port;
  Outgoing: out event data port;
end sensor;
system sensornet
end sensornet;
system implementation sensornet.impl
subcomponents
  sensorarray; device sensor [10][10];
connections
  rows: port sensorarray.outgoing -> sensorarray.incooming
        { Connection_Pattern => ((One_To_One, Next));};
end sensornet.impl;
```

9.2.4 Port Communication Timing

(30) The content of incoming data, event, or event data ports is frozen for the duration of a thread execution, i.e., the port variable content as it is accessible to the component application code is not affected by the arrival of new data, events or event data. By default the input is frozen at the time of the dispatch, i.e., at the time when `t = 0`. If the `Input_Time` property is specified it is determined by the property value as specified in Section (14). Any input arriving after the input time becomes available at the next input time. This may be the input time for the next dispatch, or the next input time in the same dispatch, if multiple input time values are specified.

(31) The output is transferred to other components at completion time, i.e., `c` = $c_{complete}$, or as specified by the value of the `Output_Time` property (see Section (14)). Output may be sent multiple times during the same dispatch; this is specified by multiple output time values.

(32) Event and event data ports may trigger the dispatch of a `Sporadic`, `Aperiodic`, or `Timed` thread as specified in Section 5.4.2. The content of data, event, and event data ports is processed at the dispatch rate. In case of event and event data ports, the input is the queued set of items at `Input_Time`; each arrived event or event data is only processed once, and items can be processed one per dispatch or in batches.

(33) Arrival of events on event ports can also trigger a mode switch if the event port is named in a mode transition originating in the current mode (see Section 12). Events that trigger mode transitions are not queued at event ports.

(34) In case of incoming data ports, the input is the most recently arrived value at input time. This may be the same as the value at the previous dispatch. The dispatch rate determines the rate at which a data stream is sampled.

(35) An incoming data stream may be sampled periodically by a periodic thread, or it may be sampled at the rate at which `Aperiodic`, `Sporadic`, `Hybrid`, and `Timed` threads are dispatched. In this case the sampling thread samples the data stream at its dispatch rate independent of the dispatch and completion of the source thread. If the incoming port is a data port the most recent value is available. If the incoming port is an event port or event data port the content of the port queue may be sampled.

(36) The actual transfer of data to data ports as ultimate destination is affected by the `Transfer_Type` property, which specifies whether the ultimate source or ultimate destination initiate the transfer, with the default being the ultimate source.

(37) The actual transfer of event data and events is affected by the `Fan_Out_Policy` property of ports with multiple outgoing connection declarations (see Section 9.2.6).

9.2.5 Sampled, Immediate, and Delayed Data Port Communication

(38) The source of a data stream may be a data or event data port of a periodic thread or device. When this data stream is sampled by a periodic thread or device, sampling, oversampling, and undersampling may occur non-deterministically due concurrency and preemption. This can lead latency jitter in the data and instability in control system behavior. AADL supports deterministic sampling of data streams between a periodic source and destination thread by specifying immediate and delayed timing of port connections.

(39) Sampled data port communication occurs when a periodic destination thread or device with an incoming data port samples a data stream. In a sampling semantic connection the recipient samples the output of the sender at dispatch time or as specified by the *Input_Time* property of the recipient port. Since this sampling occurs independent of the dispatch and completion of the source thread the data stream is sampled non-deterministically as illustrated in Figure 16. It shows two threads executing concurrently at 10 Hz and 20 Hz on different processor cores. The output of the first dispatch of Thread 1 is received by the second dispatch of Thread 2. The output of the second dispatch of Thread 1 is received by the fifth dispatch of Thread 2, since the fourth dispatch of T2 occurs before the second dispatch of T1 completes.

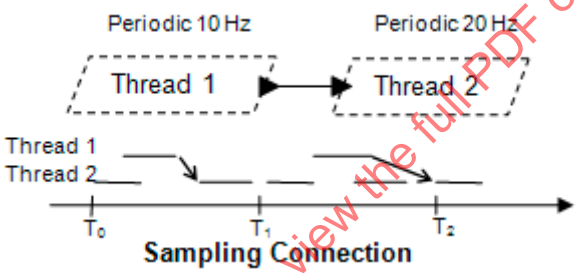


Figure 16 Sampling Data Port Connection

(40) Port connections can also be declared to be deterministic, i.e., they are declared to have an *immediate* or *delayed* `Timing` property value. If the ultimate source and destination of a semantic connection are periodic threads or devices and the ultimate destination is a data port, then a semantic connection with an immediate or delayed `Timing` property value imposes special communication timing semantics.

(41) In an immediate semantic connection the sender always communicates with the receiver mid-frame, i.e., in the same dispatch frame. In this case the receiver, when dispatched at the same time or at the first dispatch after the sender dispatch, waits for the sender to complete its execution. The scheduler must ensure that the execution of the receiver is aligned with the completion of the sender.

(42) In a delayed semantic connections the sender always communicates with the recipient phase-delayed, i.e., in the next dispatch frame of the recipient after the deadline of the sender. In this case, the send and receipt times are specified in terms of clock time, thus, ensuring determinism. The communication mechanism takes care of delaying the communication and the scheduler is unaware of this delay.

(43) Deterministic sampling is ensured within a synchronization domain. In the case of asynchronous systems, deterministic sampling is not guaranteed unless appropriate protocols are provided by the runtime system to ensure logically coordinated sampling that accommodates time drift across synchronization domains.

(44) Immediate and delayed connection timing are illustrated in Figure 17. Thread 1 and Thread 2 are two periodic threads executing at a rate of 10Hz, i.e., they are logically dispatched every 100 ms. Immediate connection timing semantics are shown on the left of the figure, and delayed are shown on the right of the figure.

(45) For immediate connection timing the actual start of execution of the receiving thread (Thread 2) will be delayed if its dispatch occurs at the same time or after the dispatch of the sending thread (Thread 1) and before execution completion of the sending thread. At the actual start time the sending thread **out** port data value is transferred into the **in** port of the receiving thread. For example, if Thread 2 executes at twice the rate of Thread 1, then the execution of Thread 2 will be delayed every time the two periods align to receive the data at completion of Thread 1. Every other time Thread 2 will start executing at its dispatch time with the old value in its data port.

(46) For delayed connection timing, the data is not made available to the recipient until the deadline of the source thread. The data is available at the destination port at the next dispatch of the destination thread that occurs at or after the source thread deadline. If the source deadline and the destination dispatch occur at the same logical time instant, the transmission is considered to occur within the same time instant. The output of Thread 1 is made available to Thread 2 at the beginning of its next dispatch. Thread 1 producing a new output at the next dispatch does not affect this value.

Note: The data transfer of a delayed connection may be initiated at the time of send, but the runtime system must ensure through double buffering that the data is not moved to the in port variable for receipt by the recipient until after the deadline of the sender.

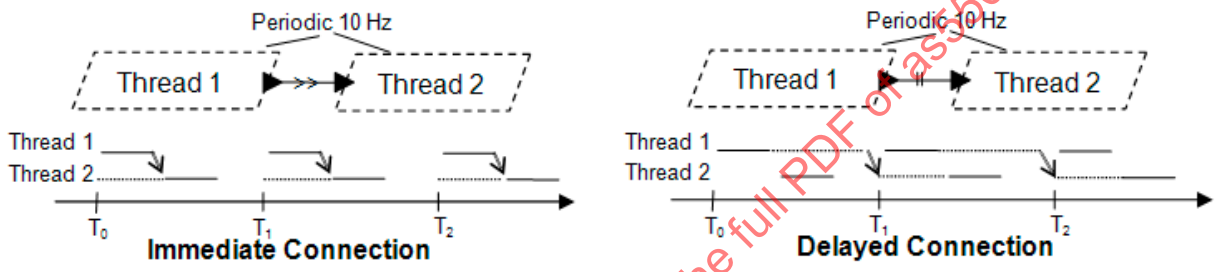


Figure 17 Timing of Immediate & Delayed Data Connections

(47) If the connection declarations that comprise a semantic port connection have an explicit `Timing` property association, then the value must be the same. The property is only interpreted if the source and destination are periodic and the destination feature is a data port.

(48) For immediate data port connections data transfer occurs at completion time of the sender ($c_{source} = c_{complete,source}$) and the receiver execution start time is delayed until the sender completes ($c_{destination} = 0 \wedge c_{source} \leq c_{complete,source}$). Both the source and destination must complete their execution by the deadline of the destination, i.e., ($c_{source} = c_{complete,source} \wedge c_{destination} = c_{complete,destination} \wedge t_{destination} \leq Deadline_{destination}$). This rule is transitive for sequences of immediate semantic connections. Note that the output may occur at a time before completion as specified by the `Output_Time` property. In this case, $c_{complete,source}$ becomes $c_{output\_time,source}$.

(49) For delayed data port connections data transmission is initiated at the deadline of the source component ($t_{source} = Deadline_{source}$, i.e., the output time of the source data port is `Deadline_Time`). The input time of the receiving component port is the `Dispatch_Time`, i.e., the data is received at the next dispatch of the receiving component following or equal to the source deadline.

(50) For immediate connections the `Input_Time` is assumed to be start time with zero offset and any other specified time is ignored. The `Output_Time` is assumed to be completion time or must be specified as a single output time value.

(51) For delayed connections the `Input_Time` is assumed to be dispatch time and `Output_Time` is assumed to be deadline.

(52) If multiple transmissions occur for a data port connection from the source thread before the dispatch of the destination thread, then only the most recently transmitted data is available in the destination port. In other words, the destination thread *undersamples* the transmitted data. In the case of two connected periodic threads, this situation occurs when the source thread has a shorter period than the destination thread. In the case of a periodic thread connected to an aperiodic thread, this situation occurs if the aperiodic thread receives two dispatch events with an inter-arrival time larger than the period of the source thread. In the case of an aperiodic thread connected to a periodic thread, this situation occurs if the aperiodic thread has two successive completion times less than the period of the periodic thread.

(53) If no transmission occurs on an in data port between two dispatches of the destination thread, then the thread receives the same data again, resulting in *oversampling* of the transmitted data. A status indicator is accessible to the source text of the thread as part of the port variable to determine whether the data is fresh. This allows a receiving thread to determine whether a connection source is providing data at the expected rate or fails to provide data.

9.2.6 Semantic Port Connections and Port Queues

(54) If the ultimate destination of a semantic port connection is an event port or event data port, then this port has by default a queue of size one. The size of this queue can be changed by explicitly with the `Queue_Size` property.

(55) Ports that are part of the sequence of connection declarations of a semantic connection can have a fan-out policy called of `OnDemand`. Such a fan-out policy results in queuing of data and events for retrieval by the ultimate destination (see Section 8.3.3).

(56) This means that the output from the ultimate source of a semantic connection is passed into the queue of the first port with an `OnDemand` fan-out policy and more than one outgoing connection; in the case of a single outgoing connection the output can be passed on without queuing.

(57) If the port of the ultimate destination of a semantic connection does not receive input from a semantic connection, then it services the queue of the last port in its connection declaration sequence with an `OnDemand` fan-out policy, more than one outgoing connection, and at least one semantic connection that provides input into its queue. If a port with an `OnDemand` fan-out policy only receives input from one port with an `OnDemand` fan-out policy, then that port is the port to be serviced.

(58) If the ultimate destination is a thread or device with multiple ports that can trigger a dispatch, then they are serviced according to the rules specified for the `Urgency` property in Section 8.3.3.

(59) More complex queue processing patterns can occur if a port with `OnDemand` fan-out policy and more than one outgoing connection has input from an ultimate source of a semantic connection and from another port with `OnDemand` fan-out, or from multiple ports with `OnDemand` fan-out, In the former case, the other port queue is only serviced if the original queue is empty. In the latter case one of those port queues can be chosen according to a fan-in prioritization of the port.

(60) If the ultimate destination of a semantic port connection is a mode transition, then the arrival of an event or event data at the port queue results in immediately passing of a mode transition trigger event to the mode transition, in addition to its queuing in the port queue for the purpose of ultimate destination dispatch and input (see also Section 12).

Processing Requirements and Permissions

(61) The temporal semantics for port connections define several cases in which the transmission initiation and completion times are identical. While it is not possible to perform a transmission instantaneously in a actual system, a method of implementing systems must supply a thread execution schedule that preserves the temporal and logical semantics of the model. In some cases, this may result in a system where the actual sending thread completion time occurs before the logical departure time of the transmission. Similarly, the actual receiving thread may begin its execution after the logical arrival of the transmission. Such an execution model is acceptable if the observed causal order is identical to that of the logical semantic model and all timing requirements specified in all property associations are satisfied.

(62) For port connections between periodic threads, the standard semantics and default property associations result in undersampling when the period of the sending thread is less than the period of the receiving thread. Oversampling occurs when the period of the sending thread is greater than the period of the receiving thread. A method of implementing systems is permitted to provide an optimization which may eliminate any physical transfers of data that can be guaranteed to be overwritten, or that can be guaranteed to be identical to previously transferred values. Error-free transmission may be assumed when performing such an optimization.

(63) A method of building systems must include a runtime capability in every system to detect an erroneous or failed transmission over a data connection between periodic threads to the degree of assurance required by an application. A method of building systems must include a runtime capability in every system to report a failure to perform a transmission by a sending periodic thread to all connected recipients. A method of building systems must include a runtime capability in every system to detect data errors in arriving transmissions over any connection to the degree of assurance required by an application. The source language annex to this standard specifies the application program interface used to obtain error information about transmissions. A method of building systems may define additional error semantics and error detection mechanisms and associated application programming interfaces.

(64) Port connections can have shared data components as source. This requires the runtime system to monitor write operations to the data component. A method of building systems may choose to not support this capability.

(65) Deterministic communication expressed by immediate and delayed connections must be guaranteed by the method of implementation within a synchronization domain. Even if the transmission is initiated and completed by explicit send and receive service calls in the source text of the sending and receiving thread, the send and receive order of the two communicating threads must be assured. A method of implementation may choose not to support immediate and delayed connections across synchronization domains.

NOTES:

All data values that arrive at the data ports of a receiving thread are immediately transferred at the logical arrival time into the storage resources associated with those features in the source text and binary image associated with that thread. A consequence of the semantic rules for data connections is that the logical arrival time of a data value to a data port contained in a thread always occurs either when that thread is dispatchable or during an interval of time between a dispatch event and a delayed start of execution, e.g., due to an immediate connection. That is, data values will never be transferred into a thread's data ports between the time it starts executing and the time it completes executing and suspends awaiting a new dispatch unless such an input time is specified through the `Input_Time` property.

Arriving event and event data values may be queued in accordance with the queuing rules defined in the port features section. A consequence of the semantic rules for event and event data connections is that there will be exactly one dispatch of a receiving thread for each arriving event or event data value that is not lost due to queue overflow, and event data values will never be transferred into a thread between the time it starts executing and the time it completes and suspends awaiting a new dispatch.

9.3 Parameter Connections

(1) Parameter connections represent flow of data between the parameters of a sequence of subprogram calls in a thread. Parameter connections may be declared from an incoming data or event data port of the containing thread to an incoming parameter of a subprogram call. Parameter connections also specify connections from an incoming parameter of the containing subprogram to an incoming parameter of a subprogram call, from an outgoing parameter of a subprogram call to an outgoing parameter of the containing subprogram, and from an outgoing parameter of a subprogram call to an incoming parameter of a subprogram call or to an outgoing data or event data port of the containing thread. In addition, parameters may be connected to data components (either to data subcomponents or data access features) to represent data flowing to and from static and local variables. In summary, the parameter connections follow the containment hierarchy of subprogram calls nested in other subprograms. This is illustrated in Figure 18. The call order is from left to right.

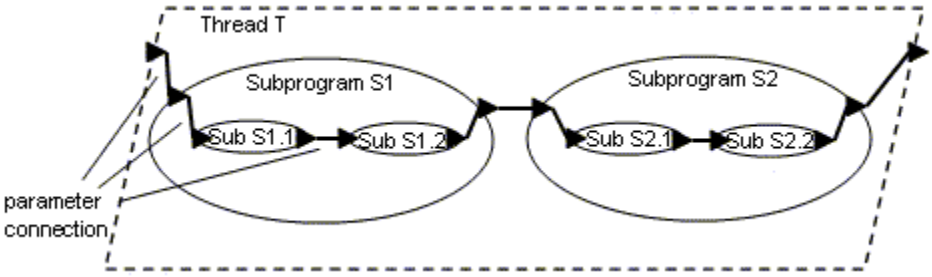


Figure 18 Parameter Connections

(2) For parameter connections, data transfer occurs at the time of the subprogram call and call return. In the case of subprogram calls to remote subprograms, the data is first transferred to a local proxy and from there passed to the remote subprogram.

Syntax

```
parameter_connection ::=
    parameter source_parameter_reference
        directional_connection_symbol destination_parameter_reference

parameter_connection_refinement ::=
    parameter

parameter_reference ::=
        -- parameter in the thread or subprogram type
      component_type_parameter_identifier [ . parameter_identifier ]
    |
        -- parameter in another subprogram call
      subprogram_call_identifier . parameter_identifier
    |
        -- data or event data port in the thread type
        -- or an element of that port's data
      component_type_port_identifier [ . data_subcomponent_identifier ]
    |
        -- data subcomponent in the thread or subprogram
      data_subcomponent_identifier
    |
        -- requires data access in the thread or subprogram type
      requires_data_access_identifier
    |
        -- data access element in a feature group of the component type
     component_type_feature_group_identifier . element_data_access_identifier
    |
        -- port or parameter element in a feature group of the component type
      component_type_feature_group_identifier [ . element_port_identifier ]
```

Naming Rules

(N1) The connection identifier in a parameter connection refinement declaration must refer to a named parameter or feature connection declared in an ancestor component implementation.

(N2) A source (destination) reference in a parameter connection declaration must reference a parameter of a preceding (succeeding) subprogram call, a parameter declared in the component type of the containing subprogram, a data port or event data port declared in the component type of the enclosing thread, a data port or event data port that is an element of a feature group in the component type of the enclosing thread, a data subcomponent, or a requires data access to a data component.

Legality Rules

(L1) The source of a parameter connection must be an incoming data or event data port of the containing thread, an incoming parameter of the containing subprogram, or a data subcomponent or requires data access with read-only or read-write access rights, or an outgoing parameter of a previous subprogram call.

(L2) The following source/destination pairs are acceptable for parameter connection declarations:

threadport -> call.parameter threadfeaturegroupport -> call.parameter	requiresdataaccess -> call.parameter featuregrouprequiresdataaccess -> call.parameter
call.parameter -> threadport call.parameter -> threadfeaturegroupport call.parameter -> threadincompletefeaturegroup	call.parameter -> requiresdataaccess call.parameter -> featuregrouprequiresdataaccess call.parameter -> threadincompletefeaturegroup
call.parameter -> call.parameter	datasubcomponent -> call.parameter
enclosingcall.parameter -> containedcall.parameter	call.parameter -> datasubcomponent
containedcall.parameter -> enclosingcall.parameter	

(L3) A parameter cannot be the destination feature reference of more than one parameter connection declaration unless the source feature reference of each parameter connection declaration is contained in a different mode. In this case, the restriction applies for each mode.

(L4) The data classifier of the source and destination must match. The matching rules as specified by the `Classifier_Matching_Rule` property apply (see Section 9.2 (L14)). By default the data classifiers must be match.

Semantics

(3) Parameter connections represent sequential flow of data through subprogram parameters in a sequence of subprogram calls being executed by a thread.

(4) Parameter connections are restricted to 1-n connectivity, i.e., a data port or parameter can have multiple outgoing connections, but only one incoming connection.

(5) If a subprogram has an **in out** parameter, this parameter may be the destination of an incoming parameter connection and the source of outgoing parameter connections.

(6) Parameter connections follow the call sequence order of subprogram calls. In other words, parameter connection sources must be preceding subprogram calls, and parameter connection destinations must be successor subprogram calls.

(7) The optional `in_modes` subclause specifies what modes the parameter connection is part of. The detailed semantics of this subclause are defined in Section 12.

Examples

NOTE: An example of parameter connections can be found in Section 5.2.

9.4 Access Connections

(1) Access connections represent access to shared data components by concurrently executing threads or by subprograms executing within thread. They also denote communication between processors, memory, and devices by accessing a shared bus. Finally, they represent a subprogram call to a specific instance of a subprogram or subprogram group. If the subprogram or subprogram group resides in a different thread, then the call is treated as a remote call. Access connections for subprograms and buses are bidirectional. Data and bus access connections may be bidirectional or directional. If they are directional they indicate write or read access to the data component or bus component.

(2) A *semantic access connection* is defined by a sequence of one or more individual access connection declarations that follow the component containment hierarchy from an *ultimate source* to an *ultimate destination*.

(3) In the case of bidirectional connections either the subcomponent being accessed or the feature that requires access can be the ultimate source or destination. In the case of directional data access connections the ultimate soruce is the source of the data flow, i.e., the data component in the case of a read access and the ultimate destination in the case of a write access. In the case of partial AADL models, the ultimate source or destination may be a provides access feature of a component without subcomponents.

(4) Figure 19 illustrates a semantic data connection from the data component D to thread Q. The access connection is bidirectional, thus, the data component could be the ultimate destination as well.

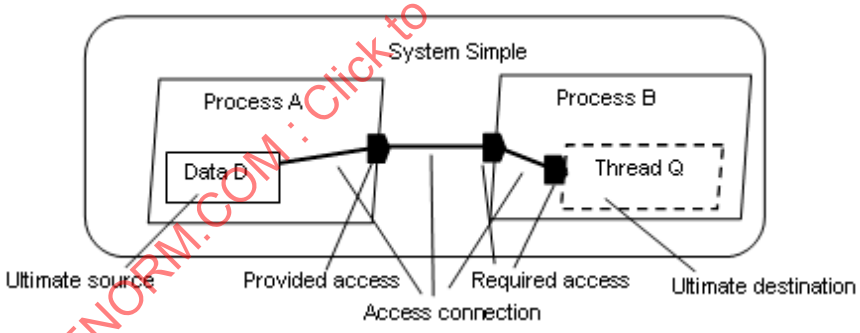


Figure 19 Semantic Access Connection For Data Components

(5) The flow of data of a semantic data or bus access connection is determined by the `Access_Right` property on the shared component or the access feature. In the case of a directional access connection and the connection direction must be consistent with the access rights.

Syntax

```
access_connection ::=
    [ bus | subprogram | subprogram group | data ] access
     access_provider_reference  connection_symbol access_requirer_reference


access_connection_refinement ::=
      [ bus | subprogram | subprogram group | data ] access
```

```
access_reference ::=
        -- requires or provides access feature in the component type

      requires_access_identifier |  provides_access_identifier
    |
        -- requires or provides access feature a feature group of the component type
      feature_group_identifier [ . requires_access_identifier ]
    |
      feature_group_identifier [ . provides_access_identifier ]
    |
       -- provides or requires access in a subcomponent
      subcomponent_identifier . provides_access_identifier
    |
      subcomponent_identifier . requires_access_identifier
    |
        -- data, subprogram, subprogram group or bus being accessed
      data_subprogram_subprogam_group_or_bus_subcomponent_identifier
    |
        -- subprogram a processor being accessed
      processor . provides_subprogram_access_identifier
```

Naming Rules

(N1) The connection identifier in an access connection refinement declaration must refer to a named access or feature connection declared in an ancestor component implementation.

(N2) A provider reference in an access connection declaration must reference a provides access feature of a subcomponent, a requires access feature in the component type of the containing component, a requires access feature in a feature group of the containing component type, an incomplete feature group of the containing component type, or a data or bus subcomponent. A requirer reference in an access connection declaration must reference a requires access feature of a subcomponent, a provides access feature in the component type of the containing component, a provides access feature in a feature group of the containing component type, or an incomplete feature group of the containing component type.

Legality Rules

(L1) The category of the source and the destination of a access connection declaration must be the same, i.e., they must both be **data**, **bus**, **subprogram**, or **subprogram group**, or their respective access feature.

(L2) In the case of a bidirectional semantic access connection either the provider of a component or the requirer may be the ultimate source. In the case of a directional semantic access connection the component being accessed must be the ultimate source for read access from the component, and the ultimate destination for write access to the component. In a partial AADL model the ultimate source or destination may be a provides access feature of a component instead of the subcomponent.

(L3) For directional access connections between data or bus access features, the direction declared for the access connection must be compatible with the direction of the read or write access based on the access rights.

The **provides** and **requires** indicators of source and destination access features must satisfy the following rules:

(L4) If the access connection declaration represents an access connection between access features of sibling components, then the source must be a **provides** access, and the destination must be a **requires** access.

(L5) If the access connection declaration represents a feature mapping up the containment hierarchy, then the source and destination must both be a **provides** access; or the source must be a data, subprogram, or bus subcomponent, and the destination a **provides** access or an incomplete feature group.

(L6) If the access connection declaration represents a feature mapping down the containment hierarchy, then the source and destination must both be a **requires** access; or the source must be a data, subprogram, or bus subcomponent or an incomplete feature group, and the destination a **requires** access.

(L7) A **requires** access cannot be the source or destination feature reference of more than one access connection declaration unless the source feature reference(s) of each access connection declaration is (are) contained in a different mode. In this case, the restriction applies for each mode.

(L8) For access connections the classifier of the provider access must match to the classifier of the requires access according to the `Classifier_Matching_Rules` property. By default the classifiers must be the same (see Section 9.1).

(L9) If more than one access feature in a semantic access connection has an `Access_Right` property association, then the resulting property values must be compatible. This means that the provider must provide `read-only` or `read-write` access if the requirer specifies `read-only`. Similarly, the provider must provide `write-only` or `read-write` access if the requirer specifies `write-only`. The provider must provide `read-write` access if the requirer specifies `read-write`. Finally, the provider must provide `by-method` access if the requirer specifies `by-method` access.

(L10) The category of the access connection source and destination must be identical. If the component category is specified as part of the connection declaration, then it must be identical to that of the source and destination. **Bus**, **data**, **subprogram**, and **subprogram group** are acceptable categories.

Standard Properties

```
Connection_Pattern: list of Supported_Connection_Patterns
Connection_Set: list of Connection_Pair
```

Semantics

(6) A data access connection represents access to a shared data component by concurrently executing threads or by subprograms executing within thread. A subprogram access or subprogram group access connection represents access to subprogram code or a library that calls can be made to. The call may be a remote call. A bus access connection represents communication between processors, memory, and devices by accessing a shared bus.

(7) Access connections are restricted to 1-n connectivity, i.e., a data, subprogram, or bus component can have multiple outgoing access connections, but a **requires** access feature can only have one connection. The restriction of one connection applies to each mode, i.e., different incoming access connections may exist in different modes, but not in the same mode.

(8) The optional `in_modes` subclause specifies what modes the access connection is part of. The detailed semantics of this subclause are defined in Section 12.

(9) The actual data flow through data and bus access connections is determined by the value of the `Access_Right` property on the shared data or bus component and their provides and requires access declarations. `Read` means flow of data from the shared component to the component requiring access, and `write` means flow of data from the component requiring access to the shared component. The desired data flow can also be indicated by specifying the direction in the access connection declaration.

(10) The connection patterns for component arrays described in Section 9.2.3 apply to access connections as well.

9.5 Feature Group Connections

(1) Feature group connections represent a collection of connections between groups of features of different components.

Syntax

```
-- connection between feature groups of two subcomponents or between
-- a feature group of a subcomponent and a feature group in the component type
feature_group_connection ::=
     feature group source_feature_group_reference
        bidirectional_connection_symbol destination_feature_group_reference

-- A feature group refinement can only add properties
-- The source and destination of the connection does not have to be repeated
feature_group_connection_refinement ::=
        feature group

feature_group_reference ::=
        -- feature group in the component type
      component_type_feature_group_identifier
    |
        -- feature group in a subcomponent
      subcomponent_identifier . feature_group_identifier
    |
        -- feature group element in a feature group of the component type
      component_type_feature_group_identifier .
          element_feature_group_identifier
```

Naming Rules

(N1) The connection identifier in a feature group connection refinement declaration must refer to a feature group named connection declared in an ancestor component implementation.

(N2) A source or destination reference in a feature group connection declaration must reference a feature group declared in the component type, a feature group of one of the subcomponents, or feature group that is an element of a feature group in the component type. The subcomponent reference may also consist of a reference on a subcomponent array.

Legality Rules

(L1) If the feature group connection declaration represents a component connection between sibling components, the feature group types must be complements. This may be indicated by both feature group declarations referring to the same feature group type and one feature group declared as **inverse of**, by the feature group type of one feature group being declared as the **inverse of** the feature group type of the other feature group, or by the two referenced feature group types meeting the complement requirements as defined in Section 8.2.

(L2) The `Classifier_Matching_Rule` property specifies the rule to be applied to match the feature group classifier of a connection source to that of a connection destination.

(L3) The following rules are supported for feature group connection declarations that represent a connection up or down the containment hierarchy:

- `Classifier_Match`: The source feature group type must be identical to the feature group type of the destination. This is the default rule.
- `Equivalence`: An indication that the two classifiers of a connection are considered to match if they are listed in the `Supported_Classifier_Equivalence_Matches` property. Matching feature group types are specified by the `Supported_Classifier_Equivalence_Matches` property with pairs of classifier values representing acceptable matches. Either element of the pair can be the source or destination classifier. Equivalence is intended to be used when the feature group types are considered to be identical, i.e., their elements match. The `Supported_Classifier_Equivalence_Matches` property is declared globally as a property constant.
- `Subset`: An indication that the two classifiers of a connection are considered to match if the outer feature group has outcoming features that are a subset of outgoing features of the inner feature group, and if the inner feature group has incoming features that are a subset of incoming features of the outer feature group. The pairs of features are expected to have the same name.

(L4) The following rules are supported for feature group connection declarations that represent a connection between two subcomponents, i.e., sibling component:

- `Classifier_Match`: The source feature group type must be the complement of the feature group type of the destination. This is the default rule.
- `Complement`: An indication that the two classifiers of a connection are considered to complement if they are listed in the `Supported_Classifier_Complement_Matches` property. Matching feature group types are specified by the `Supported_Classifier_Complement_Matches` property with pairs of classifier values representing acceptable matches. Either element of the pair can be the source or destination classifier. Complement is intended to be used when the feature group types are considered to be identical, i.e., their elements match. The `Supported_Classifier_Complement_Matches` property is declared globally as a property constant.
- `Subset`: An indication that the two classifiers of a connection are considered to match if each has incoming features that are a subset of outgoing features of the other. The pairs of features are expected to have the same name.

A feature group may have a direction declared; otherwise it is considered bidirectional. The direction declared for the destination feature group of a feature group connection declaration must be compatible with the direction declared for the source feature group as defined by the following rules:

(L5) If the feature group connection declaration represents a connection between feature group of sibling components, then the source must be an outgoing feature group and the destination must be an incoming feature group.

(L6) If the feature group connection declaration represents a connection between feature groups up the containment hierarchy, then the source and destination must both be an outgoing feature group.

(L7) If the feature group connection declaration represents a connection between feature groups down the containment hierarchy, then the source and destination must both be an incoming feature group.

(L8) A feature group connection must be bidirectional or be consistent with the direction of the source and destination feature.

Standard Properties

```
Connection_Pattern: list of Supported_Connection_Patterns

Connection_Set: list of Connection_Pair

Transmission_Type: enumeration ( push, pull )

Allowed_Connection_Binding_Class:
```

```
   inherit list of classifier(processor, virtual processor, bus, virtual bus, device,
memory)

Allowed_Connection_Binding: inherit list of reference (processor, virtual processor, bus,
virtual bus, device, memory)

Not_Collocated: record (

    Targets: list of reference (data, thread, process, system, connection);

    Location: classifier ( processor, memory, bus, system ); )

Actual_Connection_Binding: inherit list of reference (processor, virtual processor, bus,
virtual bus, device, memory)
```

Semantics

(2) These connections represent a collection of *semantic connections* between the individual features in each group. A semantic feature connection is determined by a sequence of one or more individual connection declarations that follow the component containment hierarchy in a fully instantiated system from an *ultimate source* to an *ultimate destination*. As the connection declarations follow the component containment hierarchy they may involve the aggregation of features into feature groups up the hierarchy and decomposition into individual features down the hierarchy.

(3) Feature groups and feature group connections may impose a restriction on the direction of the collection of features and connections the represent.

Processing Requirements and Permissions

(4) When an AADL model with a thread architecture and fully detailed feature groups is instantiated, each semantic connection of feature group elements is included in the instance model. If the feature groups of such a model are incomplete a connection instance between the feature group instances is created. If the AADL model is not fully detailed to the thread level, connection instances are created between the leaf components in the system hierarchy.

Examples

```
package Example3

-- A simple example showing a system with two processes and threads.

data Alpha_Type

properties

   Source_Data_Size => 256 Bytes;

end Alpha_Type;


feature group xfer_plug

features

   Alpha : out data port Alpha_Type;

   Beta : in data port Alpha_Type;

end xfer_plug;


feature group xfer_socket

   inverse of xfer_plug

end xfer_socket;
```

```
thread P
features
   Data_Source : out data port Alpha_Type;
end P;

thread implementation P.Impl
properties
   Dispatch_Protocol=>Periodic;
   Period=> 10 ms;
end P.Impl;

process A
features
   Produce : feature group xfer_plug;
end A;

process implementation A.Impl
subcomponents
   Producer : thread P.Impl;
   Result_Consumer : thread Q.Impl;
connections
  port Producer.Data_Source -> Produce.Alpha;
  port Produce.Beta -> Result_Consumer.Data_Sink;
end A.Impl;

thread Q
features
   Data_Sink : in data port Alpha_Type;
end Q;

thread implementation Q.Impl
properties
   Dispatch_Protocol=>Periodic;
   Period=> 10 ms;
end Q.Impl;

process B
features
```

```
   Consume : feature group xfer_socket;
end B;

process implementation B.Impl
subcomponents
   Consumer : thread Q.Impl;
   Result_Producer : thread P.Impl;
connections
  port Consume.Alpha -> Consumer.Data_Sink;
  port Result_Producer.Data_Source -> Consume.Beta;
end B.Impl;

system Simple
end Simple;

system implementation Simple.Impl
subcomponents
   pr_A : process A.Impl;
   pr_B : process B.Impl;
connections
   feature group pr_A.Produce -> pr_B.Consume;
end Simple.Impl;
end Example3;
```

10 Flows

(1) A *flow* is a logical flow of data and control through a sequence of threads, processors, devices, and port connections or data access connections. A component can have a flow specification, which specifies whether a component is a flow source, i.e., the flow starts within the component, a flow sink, i.e., the flow ends within the component, or there exists a flow path through the component, i.e., from one of its incoming ports to one of its outgoing ports.

(2) The purpose of providing the capability of specifying end-to-end flows is to support various forms of flow analysis, such as end-to-end timing and latency, reliability, numerical error propagation, Quality of Service (QoS) and resource management based on operational flows. To support such analyses, relevant properties are provided for the end-to-end flow, the flow specifications of components, and the ports involved in the flow to be analyzed. For example, to deal with end-to-end latency the end-to-end flow may have properties specifying its expected maximum latency and actual latency. In addition, ports on individual components may have flow specific properties, e.g., an **in** port property specifies the expected latency of data relative to its sensor sampling time or in terms of end-to-end latency from sensor to actuator to reflect the latency assumption embedded in its extrapolation algorithm.

(3) Flows are represented by flow specification, flow implementation, and end-to-end flow declarations.

(4) A flow specification declaration in a component type specifies an externally visible flow through a component's ports, feature groups, parameters, or data access features. The flow through a component is called a *flow path*. A flow originating in a component is called the *flow source*. A flow ending in a component is called the *flow sink*.

(5) A flow implementation declaration in a component implementation specifies how a flow specification is realized in the implementation as a sequence of flows through subcomponents along connections from the flow specification origin to the flow specification destination. This is illustrated in Figure 20. The system type S1 is declared with three ports and two flow specifications. These are the flows through system S1 that are externally visible. In the example, both flows are flow paths, i.e., they flow through the system. The ports identified by the flow specification do not have to have the same data type, nor do they have to be the same port type, i.e., one can be an event port and the other an event data port. This allows flow specifications to be used to describe logical flows of information.

(6) The system implementation for system S1 is shown on the right of Figure 20. It contains two process subcomponents P1 and P2. Each has two ports and a flow path specification as part of its process type declaration. The flow implementation of flow path F1 is shown in both graphical and textual form. It starts with port pt1, as specified in the flow specification. It then follows a sequence of connections and subcomponent flow specifications, modeled in the figure as the sequence of connection C1, subcomponent flow specification P2.F5, connection C3, subcomponent flow specification P1.F7, connection C5. The flow implementation ends with port pt2, as specified in the flow specification for F1.

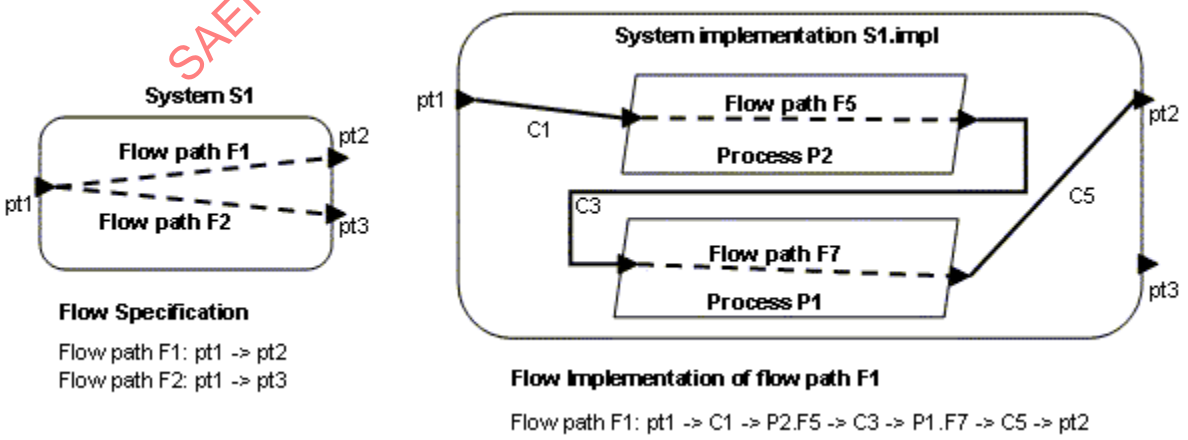


Figure 20 Flow Specification & Flow Implementation

(7) An end-to-end flow is a logical flow through a sequence of system components, i.e., threads, devices and processors. An end-to-end flow is specified by an end-to-end flow declaration. End-to-end flow declarations are declared in component implementations, typically the component implementation in the system hierarchy that is the root of all threads, data components, processors, and devices involved in an end-to-end flow. The subcomponent identified by the first subcomponent flow specification referenced in the end-to-end flow declaration contains the system component that is the starting point of the end-to-end flow. Succeeding named subcomponent flow specifications contain additional system components. In the example shown in Figure 20, the flow specification F7 of process P1 may have a flow implementation that includes flows through two threads which is not included in this view of the model.

10.1 Flow Specifications

(1) A flow specification declaration indicates that information logically flows from one of its incoming ports, parameters, or feature groups to one of its outgoing ports, parameters, or feature groups, or to and from data components via data access. The ports can be event, event data, or data ports. A flow may start within the component, called a *flow source*. A flow may end within the component, called a *flow sink*. Or a flow may go through a component from one of its **in** or **in out** ports or parameters to one of its **out** or **in out** ports or parameters, called a *flow path*. A flow may also follow read or write access to data components. In the case of feature groups, there is a flow from a feature group to its inverse.

(2) Multiple flow specifications can be defined involving the same features. For example, data coming in through an **in** feature group is processed and derived data from one of the feature group's contained ports is sent out through different **out** ports.

Syntax

```
flow_spec ::=
   flow_source_spec
   | flow_sink_spec
   | flow_path_spec

flow_spec_refinement ::=
   flow_source_spec_refinement
   | flow_sink_spec_refinement
   | flow_path_spec_refinement

flow_source_spec ::=
    defining_flow_identifier : flow source out_flow_feature_identifier
        [ { { property_association }+ } ]
        [ in_modes ] ;

flow_sink_spec ::=
    defining_flow_identifier : flow sink in_flow_feature_identifier
        [ { { property_association }+ } ]
        [ in_modes ] ;

flow_path_spec ::=
    defining_flow_identifier : flow path in_flow_feature_identifier ->
```

```
                                    out_flow_feature_identifier
        [ { { property_association }+ } ]
        [ in_modes ] ;

flow_source_spec_refinement ::=
    defining_flow_identifier :
        refined to flow source { { property_association }+ }
        [ in_modes ] ;

flow_sink_spec_refinement ::=
    defining_flow_identifier :
        refined to flow sink { { property_association }+ }
        [ in_modes ] ;

flow_path_spec_refinement ::=
    defining_flow_identifier :
        refined to flow path { { property_association }+ }
        [ in_modes ] ;

flow_feature_identifier ::=
    feature_identifier
    | feature_group_identifier
    | feature_group_identifier . feature_identifier
    | feature_group_identifier . feature_group_identifier
```

Naming Rules

(N1) The defining flow identifier of a flow specification must be unique within the interface name space of the component type.

(N2) The flow feature identifier in a flow path must refer to a port, parameter, data access feature, or feature group in the component type, or to a port, data access feature, or feature group contained in a feature group in the component type.

(N3) The defining flow identifier of a flow specification refinement must refer to a flow specification or refinement in an ancestor component type.

Legality Rules

(L1) The direction declared for the *out_flow* of a flow path specification declaration must be compatible with the direction declared for the *in_flow* as defined by the following rules:

(L2) If the *in_flow* is a port or parameter, its direction must be must be an **in** or an **in out**.

(L3) If the *out_flow* is a port or parameter, its direction must be an **out** or an **in out**.

(L4) If the *in_flow* is a data access, its access right must be must be `Read_Only` or `Read_Write`.

(L5) If the *out_flow* is a data access, its direction must be `Write_Only` or `Read_Write`.

(L6) If the *in_flow* is a feature group then it must contain at least one port or data access that satisfies the above rule, or it must have an empty set of ports.

(L7) If the *out_flow* is a feature group then it must contain at least one port or data access that satisfies the above rule, or it must have an empty set of ports.

(L8) The direction declared for the out flow of a flow source specification declaration must be the same as for out flows in flow paths.

(L9) The direction declared for the *in_flow* of a flow sink specification declaration must be the same as for *in_flow* of flow paths.

(L10) If the flow specification refers to a feature group then the feature group must contain at least one port, data access, or parameter that satisfies the above rules, or the feature group must have an empty list of features.

Standard Properties

`Latency: Time_Range`

NOTES:

These properties are examples of properties for latency and throughput analysis. Additional properties are also necessary on ports to fully support throughput analysis, such as arrival rate and data size. Appropriate properties for flow analysis may be defined by the tool vendor or user (see Section 11).

Semantics

(3) A flow specification declaration represents a logical flow originating from within a component, flowing through a component, or ending within a component.

(4) In case of a flow through a component, the component may transform the input into a different form for output. In case of data or event data port, the data type may change. Similarly the flow path may be between different port types, e.g., an event port and a data port, and between ports, parameters, data access features, and feature groups. This permits end-to-end flows to be specified as logical information flows through a system despite the fact that the information is being manipulated and its representation changed.

Examples

```
package gps
public
   data signal_data
   end signal_data;

   data position
   end position;

   data implementation position.radial
   end position.radial;

   data implementation position.cartesian
   end position.cartesian;
end gps;
```

```
package FlowExample
public
process foo
features
   Initcmd: in event port;
   Signal: in data port gps::signal_data;
   Result1: out data port gps::position.radial;
   Result2: out data port gps::position.cartesian;
   Status: out data port;
flows
   -- two flows split from the same input
   Flow1: flow path signal -> result1;
   Flow2: flow path signal -> result2;
   -- An input is consumed by process foo through its initcmd port
   Flow3: flow sink initcmd;
   -- An output is generated (produced) by process foo and made available
   -- through its port Status;
   Flow4: flow source Status;
end foo;
thread bar
features
     p1 : in data port gps::signal_data;
     p2 : out data port gps::position.radial;
     p3 : out event port;
flows
     fs1 : flow path p1 -> p2;
     fs2 : flow source p3;
end bar;

thread implementation bar.basic
end bar.basic;

thread baz
features
     p1 : in data port gps::position.radial;
     p2 : out data port gps::position.cartesian;
     reset : in event port;
flows
```

```
    fs1 : flow path p1 -> p2;
    fsink : flow sink reset;
end baz;

thread implementation baz.basic
end baz.basic;
end FlowExample;
```

10.2 Flow Implementations

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

Syntax

```
flow_implementation ::=
    ( flow_source_implementation
    | flow_sink_implementation
    | flow_path_implementation )
    [ { { property_association }+ } ]
    [ in_modes_and_transitions ] ;

flow_source_implementation ::=
    flow_identifier : flow source
      { subcomponent_flow_identifier -> connection_identifier -> }*
      out_flow_feature_identifier

flow_sink_implementation ::=
    flow_identifier : flow sink
      in_flow_feature_identifier
      { -> connection_identifier -> subcomponent_flow_identifier }*

flow_path_implementation ::=
    flow_identifier : flow path
      in_flow_feature_identifier
      [ { -> connection_identifier -> subcomponent_flow_identifier }+
         -> connection_identifier ]
```

```
      -> out_flow_feature_identifier


subcomponent_flow_identifier ::=
    ( subcomponent_identifier [ . flow_spec_identifier ] )
    | data_component_reference


data_component_reference ::=
    data_subcomponent_identifier | requires_data_access_identifier
    | provides_data_access_identifier
```

Naming Rules

(N1) The flow identifier of a flow implementation must name a flow specification in the component type. Each flow implementation must be declared at most once in each component implementation. For mode-specific flow implementations, as indicated by the `in_modes_and_transitions` subclause, a flow implementation name may appear more than once.

(N2) The *in_flow* and *out_flow* feature identifier in a flow implementation must refer to the same *in_flow* and *out_flow* feature as the flow specification it implements.

(N3) The subcomponent flow identifier of a flow implementation must name a subcomponent and optionally a flow specification in the component type of the named subcomponent, or it must name a data component in the form of a data subcomponent, provides data access, or requires data access.

(N4) In case of a flow source implementation the flow identifier of the first subcomponent must refer to a flow source or a data component.

(N5) In case of a flow sink implementation the flow identifier of the last subcomponent must refer to a flow sink or a data component.

(N6) In all other cases the subcomponent flow identifier must refer to a flow path or a data component.

(N7) The connection identifier in a flow implementation must refer to a connection in the component implementation.

Legality Rules

(L1) The source of a connection named in a flow implementation declaration must be the same as the *in_flow* feature of the flow implementation, or as the out flow feature of the directly preceding subcomponent flow specification, if present.

(L2) The destination of a connection named in a flow implementation declaration must be the same as the out flow feature of the flow implementation, or as the in_flow feature of the directly succeeding subcomponent flow specification, if present.

(L3) The *in_flow* feature of a flow implementation must be identical to the *in_flow* feature of the corresponding flow specification. The *out_flow* feature of a flow implementation must be identical to the *out_flow* feature of the corresponding flow specification.

(L4) If the component implementation provides mode-specific flow implementations, as indicated by the **`in modes`** statement, then there must be a flow implementation for each of the modes.

(L5) In case of a mode-specific flow implementation, the named connections and the subcomponents of the named flow specifications must be declared for the modes listed in the **`in modes`** statement.

(L6) Flow implementations may be declared from an *in_flow* feature directly to an *out_flow* feature.

(L7) If a connection or subcomponent named in a flow implementation is refined with an **in modes** clause in a component implementation extension, then the flow implementation has to be defined again in the component implementation extension.

(L8) Component implementation extensions may declare flow implementations for flow specifications that were already declared in the component implementation being extended.

Consistency Rules

(C1) If the component implementation has subcomponents, then a flow implementation declaration is required for each flow specification for a fully refined flow.

Standard Properties

```
Latency: Time_Range
Actual_Latency: Time_Range
```

NOTES:

These properties are examples of properties for latency and throughput analysis. Their values represent the values of the flow implementation, which must satisfy the constraints of the property values of the flow specification. The semantics of the constraint are analysis specific.

Semantics

(2) A flow implementation declaration represents the realization of a flow specification in the given component implementation. Different mode-specific flow implementations may be declared for the same flow specification.

(3) A flow path implementation starts with the port or data access named in the corresponding flow specification, passes through zero or more subcomponents, and ends with the port or data access named in the corresponding flow specification (see Figure 20). A flow source implementation ends with the port or data access named in the corresponding flow specification. A flow sink implementation starts with the port or data access named in the corresponding flow specification. A flow path implementation may specify a flow that goes directly from an *in_flow* feature to an *out_flow* feature without any connections in between.

(4) A flow implementation may refer to subcomponents without identifying a flow specification in the subcomponent. In this case, the flow specification is inferred from the destination port of the preceding connection and the source port of the succeeding connection.

(5) A flow implementation within a thread may be modeled as flow through subprogram calls via their parameters. If the call is to a subprogram via a subprogram access feature, the flow goes to the subprogram instance being called. This may be a subprogram in a different thread that is called remotely.

(6) A flow through a component may transform the input into a different form for output. In case of data, event data port, or data access, the data type may change. Similarly the flow path may be between different port types, between ports and feature groups, and between data access and ports or feature groups. This permits end-to-end flows to be specified as logical information flows through a system despite the fact that the information is being manipulated and its representation changed.

(7) The optional `in_modes_and_transitions` subclause specifies what modes the flow implementation is part of. The detailed semantics of this subclause are defined in Section 12.

(8) Flow specifications can have component implementation specific and mode specific property values. The respective property associations can be declared in the properties subclause of the component implementation.

Examples

```
-- This is an AADL fragement
-- process foo is declared in the previous section
process implementation foo.basic
subcomponents
   A: thread bar.basic;
   -- bar has a flow path fs1 from port p1 to p2
   -- bar has a flow source fs2 to p3
   B: thread baz.basic;
   -- baz has a flow path fs1 from port p1 to p2
   -- baz has a flow sink fsink in port reset
connections
   conn1: port signal -> A.p1;
   conn2: port A.p2 -> B.p1;
   conn3: port B.p2 -> result2;
   conn4: port A.p2 -> result1;
   conn6: port A.p3 -> status;
   connToThread: port initcmd -> B.reset;
flows
   Flow1: flow path
            signal -> conn1 -> A.fs1 -> conn4 -> result1;
   Flow2: flow path
            signal -> conn1 -> A.fs1 -> conn2 ->
            B.fs1 -> conn3 -> result2;
   Flow3: flow sink initcmd -> connToThread -> C.fsink;
   -- a flow source may start in a subcomponent,
   -- i.e., the first named element is a flow source
   Flow4: flow source A.fs2 -> conn6 -> status;
end foo.basic;
```


10.3 End-To-End Flows

(1) An end-to-end flow represents a logical flow of data and control from a source to a destination through a sequence of threads that process and possibly transform the data. In a complete AADL specification, the source and destination can be threads, data components, devices, and processors. In an incomplete AADL specification, the source and destination are the leaf nodes in the component hierarchy, which may be thread groups, processes, or systems. If the AADL specification includes subprogram calls, the end to end flow follows the semantic connections between threads, processors, and devices, and the parameter connections within threads.

Syntax

```
end_to_end_flow_spec ::=
    defining_end_to_end_flow_identifier : end to end flow
      start_subcomponent_flow_or_etef_identifier
      { -> connection_identifier
        -> flow_path_subcomponent_flow_or_etef_identifier }*
      -> connection_identifier -> end_subcomponent_flow_or_etef_identifier
    [ { ( property_association }+ } ]
    [ in_modes_and_transitions ] ;


end_to_end_flow_spec_refinement ::=
    defining_end_to_end_identifier :
        refined to end to end flow
        ( { { property_association }+ } [ in_modes_and_transitions ]
        | in_modes_and_transitions
        ) ;


subcomponent_flow_or_etef_identifier ::=
     subcomponent_flow_identifier | end_to_end_flow_identifier
```

Naming Rules

(N1) The defining end-to-end flow identifier of an end-to-end flow declaration must be unique within the local name space of the component implementation containing the end-to-end flow declaration.

(N2) The connection identifier in an end-to-end flow declaration must refer to a connection in the component implementation.

(N3) The subcomponent flow identifier of an end-to-end flow declaration must name an optional flow specification in the component type of the named subcomponent or to a data component in the form of a data subcomponent, provides data access, or requires data access.

(N4) The end-to-end flow identifier referenced in an end-to-end flow declaration must name an end-to-end flow in the name space of the same component implementation.

(N5) The defining identifier of an end-to-end flow refinement must refer to an end-to-end flow or refinement in an ancestor component implementation.

Legality Rules

(L1) The flow specifications identified by the *flow_path*_subcomponent_flow_identifier must be flow paths, if present.

(L2) The *start*_subcomponent_flow_identifier must refer to a flow path or a flow source, or to a data component.

(L3) The *end*_subcomponent_flow_identifier must refer to a flow path or a flow sink, or to a data component.

(L4) If an end-to-end flow is referenced in an end-to-end flow declaration, then its first and last subcomponent flow must name the same port as the preceding or succeeding connection.

(L5) In case of a mode specific end-to-end flow declarations, the named connections and the subcomponents of the named flow specifications must be declared for the modes listed in the **`in modes`** statement.

Standard Properties

Error! Reference source not found.`Actual_Latency: Time_Range`

NOTES:

These properties are examples of properties for latency and throughput analysis. The expected property values represent constraints that must be satisfied by the actual property values of the end-to-end flow. The semantics of the constraint are analysis specific.

Semantics

(2) An end-to-end flow represents a logical flow of information through a system instance. The end-to-end flow is declared in a component implementation that is the common root of all components involved in the flow. The declared end-to-end flow starts with a subcomponent flow specification, followed by zero or more connections and subcomponent flow specifications, and ends with a connection and a subcomponent flow specification. The end-to-end flow can involve active components such as threads, thread groups, processes, systems, processors, and devices, as well as passive components in the form of a data component.

(3) An end-to-end flow may refer to subcomponents without identifying a flow specification. In this case, the flow specification is inferred from the destination port of the preceding connection and the source port of the succeeding connection.

(4) An end-to-end flow may be specified as a composition of other end-to-end flows, where the last element of the predecessor end-to-end flow is connected with the first element of the successor end-to-end flow.

(5) The corresponding end-to-end flow instance is determined by expanding the flow specifications through their flow implementations. The resulting end-to-end flow instance starts from a device, processor, data component, or thread that is the leaf of the component hierarchy of the first subcomponent, follows semantic connections to intermediate threads and ends with a thread, data component, device, or processor. For incomplete models or flow implementations the components involved in the end-to-end flow instance may be thread groups, processes, or systems.

(6) If the first subcomponent flow specification is a flow path, then the end-to-end flow instance starts with the first component of the flow path expanded through its flow implementation. Similarly, if the last subcomponent flow specification is a flow path, then the end-to-end flow instance ends with the last component of the flow path expanded through its flow implementation.

(7) The optional `in_modes_and_transitions` subclause specifies what modes the end-to-end flow is part of. The detailed semantics of this subclause are defined in Section 12.

Examples

```
-- This is an AADL fragment
-- process foo is declared in the previous section
process implementation foo.basic
subcomponents
   A: thread bar.basic;
   -- bar has a flow path fs1 from p1 to p2
   -- bar has a flow source fs2 to p3
   B: thread baz.basic;
```

```
   -- baz has a flow path fs1
   -- baz has a flow sink fsink
connections
   conn1: port signal -> A.p1;
   conn2: port A.p2 -> result1;
   conn3: port B.p2 -> result2;
   conn4: port A.p2 -> B.p1;
   conn5: port A.p3 -> Status;
   conn6: port A.p3 -> B.reset;
   connToThread: port initcmd -> C.reset;
flows
   Flow1: flow path
            signal -> conn1 -> A.fs1 -> conn2 -> result1;
   Flow3: flow sink initcmd -> connToThread -> C.fsink;
   -- a flow source may start in a subcomponent,
   -- i.e., the first named element is a flow source
   Flow4: flow source A.fs2 -> connect5 -> status;
   -- an end-to-end flow from a source to a sink
   ETE1: end to end flow
           A.fs2 -> conn6 -> B.fsink;
   -- an end-to-end flow where the end points are not sources or sinks
   ETE2: end to end flow
            A.fs1 -> conn4 -> B.fs1;
end foo.basic;
```

11 Properties

(1) A property provides information about model elements, i.e., component types, component implementations, subcomponents, features, connections, flows, modes, mode transitions, subprogram calls, and packages. A property has a name, a type, and a value. The property definition declares a name for a given property along with the AADL components and functionality to which the property applies. The property type specifies the set of acceptable values for a property. Each property has a value or list of values that is associated with the named property in a given specification.

(2) A property set contains declarations of property types and property definitions that may appear in an AADL specification. The two predeclared property sets in this standard define properties and property types that are applicable to all AADL specifications. User's may define property sets that are unique to their model, project or toolset. The properties and property types that are declared in user-defined property sets are accessed using their qualified name. A property definition declaration within a property set indicates the component types, component implementations, subcomponents, features, connections, flows, modes, and subprogram calls, for which this property applies.

(3) Properties can have associated expressions that are statically typed, and evaluate to a specific value. The time at which a property expression is evaluated may depend on the property and on how a specification is processed. For example, some expressions may be evaluated immediately, some after binding decisions have been made, and some reflect runtime state information, e.g., the current mode. During analysis, all property expressions can be evaluated to known values, if necessary, by considering all possible runtime states. A given property definition may have a default expression.

11.1 Property Sets

(1) A property set defines a named group of property types, property definitions, and property constant values.

Syntax

```
property_set ::=
    property set defining_property_set_identifier is
       { import_declaration }*
       { property_type_declaration
         | property_definition_declaration
         | property_constant }*
    end defining_property_set_identifier ;
```

Naming Rules

(N1) Property set defining identifiers must be unique in the global namespace.

(N2) The defining identifier following the reserved word **end** must be identical to the defining identifier following the reserved word **property set**.

(N3) Associated with every property set is a *property set namespace* that contains the defining identifiers for all property types, property definitions, and property constants declared within that property set. This means that property types, properties, and property constants with the same identifier can be declared in different property sets.

(N4) A property, property constant, or property type declared in a property set is always named by its qualified name that is the property set identifier followed by the property identifier, separated by a double colon ("::"). Predeclared properties and property types are referred to by their property identifiers without a property set qualifier.

(N5) The property set identifiers and package names listed in an `import_declaration` must exist in the global namespace.

(N6) The property set identifier of a qualified property name must be listed in an `import_declaration` of the property set or package unless it is the name of the property set that contains the qualified name.

Semantics

(2) Property definitions, property types, and property constants are organized into property sets. They are referenced by qualifying them with the property set name. This qualification is not required if the property set is one of the predeclared property sets.

(3) An `import_declaration` of a property set specifies which property sets and packages can be named in qualified references to items in other property sets.

11.1.1 Property Types

(1) A property type declaration associates an identifier with a property type. A property type denotes the set of legal values in a property association that are the result of evaluating the associated property expression.

Syntax

```
property_type_declaration ::=
     defining_property_type_identifier : type property_type ;

property_type ::=
       aadlboolean | aadlstring
     | enumeration_type | units_type
     | number_type | range_type
     | classifier_type
     | reference_type
     | record_type
     | generic_record_type

enumeration_type ::=
     enumeration ( defining_enumeration_literal_identifier
                 { , defining_enumeration_literal_identifier }*)

units_type ::=
     units units_list

units_list ::=
    ( defining_unit_identifier
        { , defining_unit_identifier => unit_identifier * numeric_literal }* )

number_type ::=
       aadlreal [ real_range ] [ units units_designator ]
     | aadlinteger [ integer_range ] [ units units_designator ]
```

```
units_designator ::=
     units_unique_property_type_identifier
     | units_list

real_range ::= real_lower_bound .. real_upper_bound

real_lower_bound ::= signed_aadlreal_or_constant

real_upper_bound ::= signed_aadlreal_or_constant

integer_range ::= integer_lower_bound .. integer_upper_bound

integer_lower_bound ::= signed_aadlinteger_or_constant

integer_upper_bound ::= signed_aadlinteger_or_constant

signed_aadlreal_or_constant ::=
    ( signed_aadlreal | [ sign ] real_property_constant_term )

signed_aadlinteger_or_constant ::=
    ( signed_aadlinteger | [ sign ] integer_property_constant_term )

sign ::= + | -

signed_aadlinteger ::=
    [ sign ] integer_literal [ unit_identifier ]

signed_aadlreal ::=
    [ sign ] real_literal [ unit_identifier ]

range_type ::=
      range of number_type
    | range of number_unique_property_type_identifier

classifier_type ::=
    classifier
    [ ( classifier_category_reference { , classifier_category_reference }* ) ]

classifier_category_reference ::=
    -- AADL or Annex meta model classifier
```

```
    classifier_qualified_meta_model_identifier

qualified_meta_model_identifier ::=
    [ { annex_identifier }** ] meta_model_class_identifier

meta_model_class_identifier ::= { identifier }+

reference_type ::=
     reference  [ ( reference_category
                  { , reference_category }* ) ]

reference_category ::=
    -- AADL or Annex meta model named element
    named_element_qualified_meta_model_identifier

unique_property_type_identifier ::=
    [ property_set_identifier :: ] property_type_identifier

property_type_designator ::=
    unique_property_type_identifier |
    property_type

record_type ::=
    record ( record_field
             { record_field }* )
record_field ::=
    defining_field_identifier : [ list of ] property_type_designator ;
```

Naming Rules

(N1) All property type defining identifiers declared within the same property set must be distinct from each other, i.e., unique within the property set namespace.

(N2) A property type is named by its property type identifier or the qualified name specified by the property set/property type identifier pair, separated by a double colon (“::”). An unqualified property type identifier must be part of the predeclared property sets.

(N3) An enumeration type introduces an enumeration namespace. The enumeration literal identifiers in the enumeration list declare an ordered list of enumeration literals. They must be unique within this namespace.

(N4) A units type introduces a units namespace. The units identifiers in the units list declare a set of units literals. They must be unique within this namespace.

(N5) The units identifier to the right of a **=>** in a units literal statement must refer to a unit identifier defined earlier in the sequence of the same units type declaration.

(N6) The *classifier meta model identifier* must refer to a class in the AADL meta model or an Annex meta model. In the case of an Annex meta model, the identifier is qualified by the annex name. Acceptable classes are listed in tabular form in Appendix C.3 and in relevant Annex standards.

(N7) The *named element meta model identifier* must refer to a class in the AADL meta model or an Annex meta model that is a subclass of the *NamedElement* class and a structural feature, in the case of the AADL core language a subclass of the *ClassifierFeature* class. In the case of an Annex meta model, the identifier is qualified by the annex name. Acceptable classes are listed in tabular form in an appendix of this standard and in relevant Annex standards.

(N8) The identifiers of the property field declarations in a **record** property type must be unique within the record declaration, i.e., the record type represents a local namespace for record field identifiers.

Legality Rules

(L1) The value of the first numeric literal that appears in a range of a `number_type` must not be greater than the value of the second numeric literal including the value's units.

(L2) Range values must always be declared with unit literals if the property requires a unit literal.

(L3) The unique property constant identifier in an integer range must represent an integer constant.

(L4) A boundless range type may be declared such that the actual range declarations have no limit on the upper and lower bound.

(L5) The unique property constant identifier in a real range must represent a real constant.

(L6) If the property requires a unit, then the unit must be specified for both lower and upper bound.

NOTES:

In the original AADL standard reserved words were used to identify the classifier category or reference category. Those names are compatible with the qualified meta model identifiers of AADL V2 with the exception of *connections*. It is now named *connection*.

Semantics

(2) A property type declaration associates an identifier with a property type.

(3) The **aadlboolean** property type represents the two values, true and false.

(4) The **aadlstring** property type represents all legal strings of the AADL.

(5) An **enumeration** property type represents an ordered list of enumeration identifiers as the set of legal values.

(6) A **units** property type represents an explicitly listed set of measurement unit identifiers as the set of legal values. The second and succeeding unit identifiers are declared with a multiplier representing the conversion factor that is applied to a preceding unit to determine the value in terms of the specified measurement unit.

(7) An **aadlreal** property type represents a real value or a real value and its measurement unit. If a units clause is present, then the type value is a pair of values, a real value and a unit. The unit may only be one of the enumeration literals specified in the units clause. If a units clause is absent, then the value is a real value. If a simple range is present, then the real value must be an element of the specified range.

(8) An **aadlinteger** property type represents an integer value or an integer value and its measurement unit. If a units clause is present, then the value is a pair of values, and unit may only be one of the enumeration literals specified in the units clause. If a units clause is absent, then the value is an integer value. If a simple range is present, then the integer value must be an element of the specified range.

(9) The **range** property type represents closed intervals of numbers. It specifies that a property of this type has a value that is a range term. The range type specifies the number type of values in the range. A property specifying a range term as its value indicates a least value called the lower bound of the interval, a greatest value called the upper bound of the interval, and optionally the difference between adjacent values called the **delta**. The delta may be unspecified, in which case the range is dense, but it is otherwise undefined whether the range is an interval of the real or the rational numbers.

(10) A **classifier** property type represents the subset of syntactically legal classifier references, whose class is a subclass of the *Classifier* meta model class and of one of the meta model classes listed in the classifier identifier list. For core AADL this is typically the meta model class that represents a component category. If the classifier identifier list is absent, all classifier references are acceptable.

(11) A **reference** property type represents the subset of syntactically legal references to those model elements, whose class is a subclass of the meta model class *Named Element* and *ClassifierFeature* or a structural element of an annex clause. A *ClassifierFeature* is a structural element that is contained in a component, such as ports, flow specifications, subcomponents, or connections. If the identifier list is absent, all model elements whose class is a subclass of *NamedElement* and *ClassifierFeature* are acceptable.

(12) A **record** type represents a group of property associations, i.e., a collection of property values, where each element of the collection of property values is accessible by name. The record fields must be explicitly declared by name, each with its own property type.

NOTES:

The classifier and reference property types support the specification of properties representing binding constraints.

Units literals are not case sensitive. For example, *mW* and *MW* represent the same units literal. Therefore, it is advisable to choose literal names such as *milliWatts* and *MegaWatts*.

Examples

```
property set mine is
  Length_Unit : type units ( mm, cm => mm * 10,
                             m => cm * 100, km => m * 1000 );
  OnOff : type aadlboolean;
  -- This type declaration references a separately declared units type
  Car_Length : type aadlreal 1.5 .. 4.5 units mine::Length_Unit ;
  -- This type declaration defines the units in place
  Speed_Range : type range of aadlreal 0 .. 250 units ( kph );
  Position : type record (
       X: aadlinteger;
       Y: aadlinteger; );
end mine;
```

11.1.2 Property Definitions

(1) All property names that appear in a property association list must be declared with property definition declarations inside a property set. Properties are typed and are defined for any named element in an AADL model.

Syntax

```
property_definition_declaration ::=
    defining_property_name_identifier:
        [ inherit ]
        ( single_valued_property | multi_valued_property )
        applies to (  property_owner { , property_owner }* ) ;

single_valued_property ::=
    property_type_designator [ => default_property_expression ]

multi_valued_property ::=
    list of property_type_designator
    [ =>
        ( [ default_property_expression { , default_property_expression }* ] )
    ]

property_owner ::=
  -- AADL or Annex meta model named element
    named_element_qualified_meta_model_identifier  |
    unique_component_type_reference
```


Naming Rules

(N1) All defining identifiers of property definitions declared within the same property set must be distinct from each other and distinct from all property type defining identifiers declared within that property set. The property set namespace contains the defining identifiers for all property definitions declared within that property set.

(N2) A property is named by its property definition identifier or the qualified name specified by the property set/property definition identifier pair, separated by a double colon ("::"). An unqualified property identifier must be part of the predeclared property sets.

(N3) The `named_element_meta_model_identifier` must identify the name of a class in the AADL meta model or an Annex meta model that is a subclass of the AADL meta model class `NamedElement`. In case of an Annex meta model, the identifier is qualified by the annex name. Acceptable classes are listed in tabular form in an appendix of this standard and in relevant Annex standards.

(N4) The `unique_component_type_reference` must identify the name of a component type in the public section of the named package.

Legality Rules

(L1) All properties are automatically defined for components of the category **abstract**, i.e., the category **abstract** is implicitly included in all **applies to** statements.

(L2) **Classifier** and **reference** property definition must not have a default value.

Semantics

(2) A property definition declaration introduces a new property that is of a specified property type, accepts a single value or a list of values, and may specify a default property expression. This property is defined for those named elements of an AADL model whose meta model name is listed after the **applies to** in the list, or those model elements that are of the specified component type. Acceptable meta model names are listed in tabular form in Appendix C.3 of this standard and in relevant Annex standards.

(3) The `Property_Owner` list may include component types qualified by a package name. In this case, the property can only be associated with components of this component type or its extensions. If the `Property_Owner` list includes `NamedElement`, then the property can be associated with all model elements.

(4) A property defined with the reserved word **inherit** indicates that if a property value cannot be determined for a component, then its value will be inherited from a containing component. The detailed rules for determining property values are described in Section 11.3.

(5) A property declared without a default value is considered undefined (see also Section 11.3). A property declared to have a list of values is considered to have an empty list if no default value is declared.

Examples

```
Value_Type: type enumeration ( estimate, benchmark, measured );
Rotation_Units: type units ( rpm );
Wheel_speed [Value_Type] : aadlinteger 0 rpm .. 5000 rpm Rotation_Units
                                          applies to ( system );
  Position : type record (
       X: aadlinteger;
       Y: aadlinteger; );
GPS_Position : Position applies to ( system );
```

11.1.3 Property Constants

(1) 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 property sets. They can be referenced in property expressions by name wherever the value itself is permissible.

Syntax

```
property_constant ::=
        single_valued_property_constant | multi_valued_property_constant

single_valued_property_constant ::=
    defining_property_constant_identifier : constant
            property_type_designator
        => constant_property_expression;

multi_valued_property_constant ::=
    defining_property_constant_identifier : constant list of
```

```
        property_type_designator
    => ( [constant_property_expression { , constant_property_expression }* ] ) ;

unique_property_constant_identifier ::=
   [ property_set_identifier :: ] property_constant_identifier
```

Naming Rules

(N1) The defining property constant identifier must be distinct from all other property constant identifiers, property definition identifiers, and property type identifiers in the namespace of the property set that contains the property constant declaration.

(N2) A property constant is named by its property constant identifier or the qualified name specified by the property set/property constant identifier pair, separated by double colon ("::"). An unqualified property constant identifier must be part of the predeclared property sets. Otherwise, the property constant identifier must appear in the property set namespace.

Legality Rules

(L1) A property constant cannot be declared for the **classifier** property type or the **reference** property type.

(L2) If a property constant declaration has more than one property expression, it must contain the reserved words **list of**.

(L3) The property type of the property constant declaration must match the property type of the constant property value.

Semantics

(2) Property constants allow integer, real, string, and other property values to be known by symbolic name and referenced by that name in property expressions. This reference is expressed by referencing the unique property constant identifier.

Examples

```
Max_Threads : constant aadlinteger => 256;
```

11.2 Predeclared Property Sets

(1) There is a standard collection of predeclared property sets named `Deployment_Properties`, `Thread_Properties`, `Timing_Properties`, `Memory_Properties`, `Programming_Properties`, and `Modeling_Properties`, which are part of every AADL specification. These property sets are listed in Appendix A .

(2) In addition, there is property set `AADL_Project` that declares a set of enumeration property types and property constants for which project-specific enumeration literals and values can be defined for different projects. This property set is part of every AADL specification. All of the property enumeration types and property constants listed in Appendix A.2 must be declared in this property set. The set of enumeration literals may vary.

Naming Rules

(N1) References to predeclared properties, property types, and property constants do not have to be qualified with the property set name. As a consequence the predeclared properties, property types, and property constants must be unique across all predeclared property sets.

NOTE: References to predeclared properties, property costants, and property types may be qualified with their property set name.

Legality Rules

(L1) The predeclared property sets other than `AADL_Project` cannot be modified.

(L2) Existing property type and property constant declarations in the `AADL_Project` property set can be modified. New declarations must not be added to the `AADL_Project` property set, but can be introduced through a separate property set declaration.

Processing Requirements and Permissions

(3) Additional property name declarations may not be inserted into the standard predeclared property sets. Separate property set declarations must be used for nonstandard property definitions.

(4) Providers of AADL processing methods may modify the standard property type declarations in `AADL_Project` to allow additional values for a specific property definitions. For example, additional enumeration identifiers beyond those listed in this standard may be added.

(5) Additional property sets may be defined. AADL tools may be defined that include support for additional property sets. Similarly, AADL specifications may be define that property associations from additional property sets.

(6) Additional property sets that may be suitable for a wide variety applications may be defined in an Annex document. AADL tools that support this Annex should include support for these additional property sets. Similarly, AADL specifications that conform to the Annex shall satisfy the requirements associated with the annex property set.

11.3 Property Associations

(1) A property association assigns a property value or list of property values with a property. This may involve evaluation of a property expressions. Property associations can be declared within component types, component implementations, subcomponents, features, connections, flows, modes, and subprogram calls, as well as their respective refinement declarations. Contained property associations permit property values to be associated with any component in the system instance hierarchy (see Section 13.1).

Syntax

```
property_association ::=
  unique_property_identifier
( => | +=> )
      [ constant ] assignment
      [ in_binding ] ;

contained_property_association ::=
  unique_property_identifier
    => [ constant ] assignment
     applies to contained_model_element_path
            { , contained_model_element_path }*
     [ in_binding ] ;

unique_property_identifier ::=
  [ property_set_identifier :: ] property_name_identifier

contained_model_element_path ::=
```

```
      ( contained_model_element { . contained_model_element }*
        [ annex_path ] )
      | annex_path


contained_model_element ::=
    named_element_identifier |
    named_element_array_selection_identifier


annex_path ::=
    annex annex_identifier {** annex_specific_path **}


annex_specific_path ::= <defined by annex>
-- It is recommended this path follows the dot-separation syntax of the component path.

assignment ::=  property_value | modal_property_value

modal_property_value ::=
    ({ property_value in_modes , }* property_value [ in_modes ] )

property_value ::= single_property_value | property_list_value

single_property_value ::= property_expression

property_list_value ::=
       ( [ property_expression { , property_expression  }* ] )

in_binding ::=
      in binding( platform_classifier_reference
                { , platform_classifier_reference }* )

platform_classifier_reference ::=
       processor_classifier_reference
       | virtual_processor_classifier_reference
       | bus_classifier_reference
       | virtual_bus_classifier_reference
       | memory_classifier_reference
```

Naming Rules

(N1) A property is named by an optional property set identifier followed by a property identifier, separated by a double colon ("::").

(N2) The property set identifier, if present, must appear in the global namespace and must be the defining identifier in a property set declaration.

(N3) The property identifier must exist in the namespace of the property set, or if the optional property set identifier is absent, in the namespace of any predeclared property set.

(N4) A property name may appear in the property association clause only if the respective AADL model element is listed in the applies to list of the property definition declaration.

(N5) The *contained model element path* identifies named model elements in the containment hierarchy, for which the property value holds. The model element with the contained property association is the root of a path.

(N6) For contained property associations declared with classifiers, e.g., a component type or component implementation, the first identifier in the reference term must appear in the local namespace of the classifier to which the property association belongs.

(N7) For contained property associations declared with model elements with a classifier reference, e.g., subcomponents, the first identifier must appear as a identifier within the local namespace of the classifier that is referenced by the model element.

(N8) Subsequent identifiers in a contained model element path must appear in the namespace of the model element identified by the preceding identifier.

(N9) The annex identifier named in a contained model element path must be that of an approved or project-specific annex.

(N10) Identifiers in an annex-specific path must appear in the namespace of the identified annex in the context of the namespace of the model element identified by the preceding identifier.

(N11) If the identifier of a contained model element path is a subcomponent array identifier, it can specify a subcomponent array as a whole, an array subset, or an individual array element.

(N12) If a property association has an **in binding** statement, then the unique platform classifier reference must be referenceable according to the **with** and **renames** declarations.

(N13) If a property association has mode-specific values, i.e., an **in modes** statement for values, then the mode must refer to a mode of the component the property is associated with, or in the case of a property association of model elements that are not components, the modes of the containing component.

(N14) A property association list must have at most one property association for the same property. In case of binding-specific property associations, there must be at most one association for each binding.

Legality Rules

(L1) The property definition named by a property association must list the class of the model element, with which the property is associated, or any of its super classes in its **applies to** clause.

(L2) If a property association is declared in the properties subclause of a package, then the property value applies to all component classifier declarations contained in the package for which the property is valid according to the **applies to** of the property definition. The property value may also be associated with the package itself, if so specified by the **applies to** of the property definition.

(L3) If a property expression list consists of a list of two or more property expressions, all of those property expressions must be of the same property type.

(L4) If the property declaration for the associated property definition does not contain the reserved words **list of**, the property value must be a single property value. If the property declaration for the associated property definition contains the reserved words **list of**, the property value can be a single property value, which is interpreted to be a list of one value.

(L5) The property association operator **+=>** must only be used if the property declaration for the associated property definition contains the reserved words **list of**.

(L6) A property association with an operator **+=>** must not have an **in modes** or **in binding** statement.

(L7) The property association operator **+=>** must not be used in contained property associations.

(L8) In a property association, the type of the evaluated property expression must match the property type of the named property.

(L9) A property value declared by a property association with the reserved word **constant** cannot be changed when the rules in the *semantics* section for determining a property value are followed.

(L10) The unique component type identifiers in the **in binding** statement must refer to component types of the categories **processor**, **virtual processor**, **bus**, **virtual bus**, or **memory.**

(L11) If a property value with an **in modes** statement is associated with a connection, flow implementation, or call sequence with an **in modes** statement, then the set of modes for which the property value applies must be contained in the set of modes for which the connection, flow implementation, or call sequence is active.

Consistency Rules

(C2) If a property association has mode-specific values, i.e. values declared with the **in modes** statement, then the modal value assignment must include a value for each mode. If the modal value assignment includes a value without the **in modes** statement, this value becomes the default for all modes without an explicit mode-specific value.

Semantics

(2) Property associations determine the property value of the model element instances in the system instance hierarchy (see Section 13.1). The property association of a classifier, subcomponent, feature, flow, connection, mode, or subprogram call and other declarative model elements determines the property value of all instances derived from the respective declaration.

(3) A property association may be declared for a package. In this case, the property value applies to the package.

(4) The value of a property is determined through evaluation of the property expression.

(5) Property associations are declared in the properties subclause of component types and component implementations. They are also declared as part of any other named declarative model element, such as features, subcomponents, connections, modes, mode transitions, etc. The property association of a component type acts as default value for all implementations, subcomponents, and instances, overwriting the default specified in the property definition. Similarly, the property association of a component implementation overwrites the value of a component type, and the subcomponent property association overwrites the value of the component implementation. The details of determining a property value are specified below.

(6) If a property association is declared with modal property values, then the same rules hold for overwriting previous property associations. Property values are modal if they are declared with an **in** modes. In this case the property value applies if one of the specified modes is active. If a property association contains both mode-specific associations and value without an **in modes** statement, then the latter specifies the value for all modes for which there is not an explicit value with an **in modes** statements. A property value without an **in modes** statement also applies when the component is inactive. If a modal property association does not specify a property value for one of the modes and there is no property value without **in modes**, then the property value is considered to be undefined.

(7) If a property association has an **in binding** statement, the property value is binding-specific. The property value applies if the binding is to one of the specified execution platform types of the categories processor, virtual processor, bus, virtual bus, or memory. If a property association list contains both binding-specific associations and an association without an in binding statement, then the latter applies to bindings to the reference processor.

(8) Contained property associations can associate property values to any named model element down the system hierarchy relative to the location of the declaration.

(9) The **applies to** clause of a contained property association may specify multiple contained model element paths. In that case the property value is associated with each of the model elements. If the property association specifying contained model elements includes an **in modes** statement, then the named modes must exist in the last component of the contained element path. If the path refers to a model element that is not a component, e.g., a feature, then the modes must exist in the containing component of that model element.

(10) A contained model element path may include the name of an array of components, or an index range of the array subset. In that case the property value is associated with each of the model elements in the array or its subset, or model elements identified by the remaining path for each of the array elements.

(11) Contained property associations can be used to record system instance specific property values for all model elements in a system instance. This permits AADL analysis tools to record system instance specific information about an actual system in a single location in an extension of the top-level system implementation. For example, a resource allocation tool can record the actual bindings of threads to processors and source text to memory through a set of contained property associations, and can keep multiple such binding configurations in different extensions of the same system implementation.

(12) The property value is determined according to the following rules:

- If a property value is not present after applying all of the rules below, it is determined by the default value of its property definition declaration. If not present in the property definition declaration, the property value is undefined.
- For classifier types, i.e., component types and feature group types and subclasses of `ClassifierType` in Annex meta models, the property value of a property is determined by its property association in the properties subclause. If not present, the property value is determined by the first ancestor classifier type in the extends hierarchy with its property association. Otherwise, it is considered not present.
- For classifier implementation, i.e., a component implementations and feature groups and subclasses of `ClassifierImplementation` in Annex meta models, the property value of a property is determined by its property association in the properties subclause. If not present, the property value is determined by the first ancestor classifier implementation in the extends hierarchy with its property association. If not present, it is determined by the property value of the classifier implementation's classifier type according to the classifier type rules.
- For modes, connections, flow sequences, or other model elements without a classifier reference, the property value of a property is determined by its property association in the element declaration. If not present and the model element is refined, then the property value is determined by a property association in the model element declaration being refined; this is done recursively along the refinement sequence. If not present and the property definition has been declared as **inherit**, it is determined by the property value of the closest containing component in the containment hierarchy of the system instance. This inherited property value must not be mode-specific. Otherwise, it is considered not present.
- For subcomponents, features and other model elements with classifier references, the property value of a property is determined by its property association in the model element declaration. If not present and the model element is refined, then the property value is determined by a property association in the model element declaration being refined; this is done recursively along the refinement sequence. If not present in the model element, it is determined by the model element's classifier reference according to the respective classifier rules described above. If not present and the property definition has been declared as **inherit**, it is determined by the property value of the closest containing component in the containment hierarchy of the system instance. This inherited property value must not be mode-specific. Otherwise, it is considered not present.

- For subprogram calls in call sequences and other model elements with classifier or feature references, the property value of a property is determined by its property association in the model element. If not present and the reference is a classifier reference, the property value is determined by the classifier according to its rules described above. If not present and the reference is a feature reference in a type, the property value is determined by the feature according to the feature rules described above. If not present and the property definition has been declared as **inherit**, it is determined by the property value of the closest containing component in the containment hierarchy of the system instance. This inherited property value must not be mode-specific. Otherwise, it is considered not present.
- For component, feature, connection, flow, mode, or other model element instances in the system instance hierarchy, the property value of a property is determined by the contained property association highest in the system instance hierarchy that references the component, feature, connection, flow, mode, or other model element. If not present, then the property value is determined by the respective subcomponent, mode, connection, feature, or other model element declaration that results in the instance according to the rules above. If not present and the property definition has been declared as **inherit**, then it is determined by the property value of the closest containing component in the containment hierarchy of the system instance. This inherited property value must not be mode-specific. Otherwise, it is undefined.

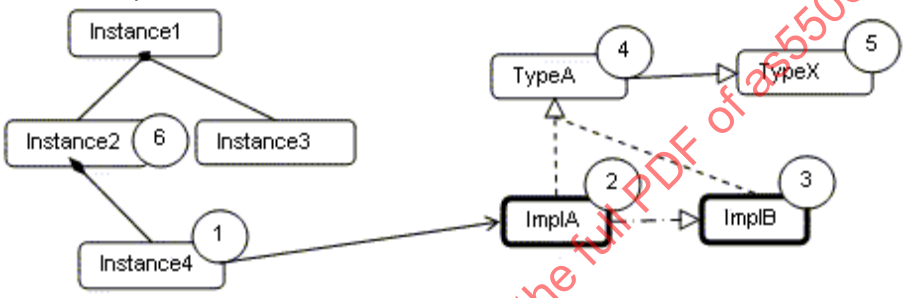


Figure 21 Property Value Determination

(13) Figure 21 illustrates the order in which the value of a property is determined. Instance4 is an element in the system instance hierarchy. The value of one of its properties is determined by first looking for a property associated with the instance itself – shown as step 1. This is specified by a contained property association. The contained property association for this instance declared in a component implementation highest in the instance hierarchy determines that value. If no instance value exists, the implementation (ImplA) of the instance is examined (step 2). If it does not exist, ancestor implementations are examined (step 3). If the property value still has not been determined, the component type is examined (step 4). If not found there, its ancestor component types are examined (step 5). If not found and the property is inherited, for subcomponents and features, the enclosing implementation is examined. Otherwise, the containing component in the component instance hierarchy is examined (step 6). Finally, the default value is considered.

(14) Two property association operators are supported. The property association operator **=>** results in a new value for the property. The property association operator **+=>** results in the addition of a value to a property value list.

(15) A property value list is evaluated by evaluating each of the property expressions, and appending the values in order. If the property expression evaluates to a list, all the list elements are appended. If the property expression evaluates to undefined, it is treated as an empty list.

(16) A property value declared by a property association with the reserved word **constant** cannot be changed. For example, if a property association is defined as **constant** for a component type, then there cannot be a property association for the same property in any component implementation of the type, nor any subcomponent, nor contained property association that applies to a component of the component type.

(17) If the property type is a **record**, then a subset of record fields may be overwritten in a property associations.

(18) Component instance property associations with specified contained subcomponent identifier sequences allow separate property values to be associated with each component instance in the containment hierarchy. In particular, it permits separate property values such as actual processor binding property values or result values from an analysis method to be associated with each component in the system instance containment hierarchy.

11.4 Property Expressions

(1) A property expression represents the value that is associated with a property through a property association. The type of the property expression must match the property type declared for the property.

Syntax

```
property_expression ::=
      boolean_term
    | real_term
    | integer_term
    | string_term
    | enumeration_term
    | unit_term
    | real_range_term
    | integer_range_term
    | property_term
    | component_classifier_term
    | reference_term
    | record_term
    | computed_term

boolean_term ::=
      boolean_value
    | boolean_property_constant_term
    | not boolean_term
    | boolean_term and boolean_term
    | boolean_term or boolean_term
    | ( boolean_term )

boolean_value ::= true | false

real_term ::=
   signed_aadlreal_or_constant

integer_term ::=
   signed_aadlinteger_or_constant

string_term ::= string_literal | string_property_constant_term

enumeration_term ::=
    enumeration_identifier | enumeration_property_constant_term
```

```
unit_term ::=
    unit_identifier | unit_property_constant_term

integer_range_term ::=
    integer_term .. integer_term [ delta integer_term ]
    | integer_range_property_constant_term

real_range_term ::=
    real_term .. real_term [ delta real_term ]
    | real_range_property_constant_term

property_term ::=
   [ property_set_identifier :: ] property_name_identifier

property_constant_term ::=
   [ property_set_identifier :: ] property_constant_identifier

component_classifier_term ::=
    classifier (
     ( unique_component_type_reference |
       unique_component_implementation_reference ) )

reference_term ::=
    reference ( contained_model_element_path )

record_term ::=
    ( record_field_identifier => property_value ;
     { record_field_identifier => property_value; }*  )

computed_term ::=
    compute ( function_identifier )
```

NOTES:

Boolean operators have the following decreasing precedence order: (), **not**, **and**, **or**.

Naming Rules

(N1) The component type identifier or component implementation name of a subcomponent classifier reference must appear namespace of the specified package.

(N2) The enumeration identifier of a property expression must have been declared in the enumeration list of the property type that is associated with the property.

(N3) For reference terms the naming rules (N5) .. (N11) in Section 11.3 are applicable in order to resolve contained model element paths.

(N4) If a **classifier** property is associated with a core AADL model element, then the classifier meta model identifier of a classifier term can only refer to a core AADL meta model class.

(N5) If a **classifier** property is associated with an AADL annex model element, then the classifier meta model identifier of a classifier term can refer to a core AADL meta model class or an annex meta model class of the same annex.

(N6) If a **reference** property is associated with a core AADL model element, then the contained model element path of a reference term can only refer to a core AADL model element.

(N7) If a **reference** property is associated with an AADL annex model element, then the contained model element path of a reference term can refer to a core AADL model element or an annex model element of the same annex.

(N8) The field identifier of a record expression must exist in the local namespace of the record type.

(N9) The function identifier of a **compute** expression must exist as function in the source text.

Legality Rules

(L1) If the base type of a property number type or range type is integer, then the numeric literals must be integers.

(L2) The type of a property named in a property term must be identical to the type of the property name in the property association.

(L3) The type of a property constant named in a property constant term must match the type of the property name in the property association.

(L4) Property references in *property_term* or *property_constant_term* of property expressions must be applicable to the model element to which the property association applies.

(L5) Property references in *property_term* or *property_constant_term* of property expressions cannot be circular. If a property has a property expression that refers to a property or property constant, then that expression evaluation cannot directly or indirectly depend on the value of the original property or property constant.

(L6) If the contained model element path of a reference term includes a subcomponent array identifier that does not identify a single element in the array, then the expression results in a list of references.

Semantics

(2) Every property expression can be evaluated to produce a value, a range of values, or a reference. It can be statically determined whether this value satisfies the property type designator of the property name in the property association. The value of the property association may evaluate undefined, if no property association or default value has been declared.

(3) *Boolean terms* are of property type **aadlboolean**. The reserved words **true** and **false** evaluate to the Boolean values true and false. The operator **not** logically negates the value of a Boolean term. Expressions containing the operators **or** and **and** are of type Boolean. They evaluate to the logical disjunction and conjunction of the values of their subexpression. Boolean operators have the following decreasing precedence order: (), **not**, **and**, **or**. Because Boolean expressions can contain property terms that reference the values of other properties, and a referenced property value could be undefined, the Boolean operators are defined to operate over the three values true, false, and undefined.

op1 and op2	**True**	**False**	**Undefined**
True	True	False	Undefined
False	False	False	False
Undefined	Undefined	False	Undefined

op1 or op2	**True**	**False**	**Undefined**
True	True	True	True
False	True	False	Undefined
Undefined	True	Undefined	Undefined

X	**Not X**
True	False
False	True
Undefined	Undefined

(4) *Number terms* evaluate to a numeric value denoted by the numeric literal, or evaluate to a pair consisting of a numeric value and the specified units identifier. A number term satisfies an **aadlinteger** property type if the numeric value is a numeric literal without decimal point or exponent. Otherwise, it satisfies the **aadlreal** property type. If specified, the units identifier must be one of the unit identifiers in the unit designator of the property type. Furthermore, the value must fall within the optionally specified range of the property type – taking into account unit conversion as necessary.

(5) *Enumeration terms* evaluate to enumeration identifiers. The **enumeration** property type of the property name is satisfied if the enumeration identifier is declared in the enumeration list of the property type.

(6) *Range terms* are of **range** property type and are represented by number terms for lower and upper range bounds plus and an optional **delta** value. Range terms evaluate to two or three numeric values that and each must satisfy the number type declared as part of the range property type. The **delta** value represents the maximum difference between two values, e.g., between two values of a data stream coming through a data port.

(7) *String terms* are of **aadlstring** property type. A string literal evaluates to the string of characters denoted by that literal.

(8) *Property terms* evaluate to the value of the referenced property. This allows one property value to be expressed in terms of another. The value of the referenced property is determined in the context of the element for which the property value is being determined. For example, the `Deadline` property has the property term `Period` as its default property expression. If this default value is not overwritten by another property association, the value of `Deadline` of a thread subcomponent is determined by evaluating the property term in the context of the thread subcomponent, i.e., the `Deadline` value is determined by the `Period` value for the thread subcomponent rather than the context of the default value declaration. The value of the referenced property may be undefined, in which case the property term evaluates to undefined.

(9) *Property constant terms* evaluate to the value of the referenced property constant. This allows one property value to be expressed symbolically in terms of a constant identifier rather than the actual value.

(10) *Component classifier terms* are of the property type **classifier**. They evaluate to a classifier reference.

(11) *Reference terms* are of **reference** property type and evaluate to a reference. This reference may be a reference to a model element in the model containment hierarchy, e.g., to a contained component, to a connection, mode, feature, or an annex model element.

(12) *Record terms* evaluate to a record value, which consists of a separate value for each named field in the record expression. Any record fields defined in a record type, for which there is no value in the record expression, the field value is determined according to the rules for property associations (see Section 11.3).

(13) *Computed terms* allow a user-defined function to be called to calculate a property value. This function is called every time the value of the property is accessed. This function may access other properties of the same and other model elements to calculate its value and it is assumed to complete its computation in finite time. A typical use of this function is to calculate the value of a property based on the value of a property of its subcomponents. It takes the property association and the model element it is associated with as parameters. The function is expected to be without side effects and to return a value that is consistent with the type of the property. The function being called is assumed to exist in a library that is supplied to methods of processing.

NOTES:

Expressions of the property type **reference** or **classifier** are provided to support the ability to refer to classifiers and model elements in the model. For example, they are used to specify bindings of application components to execution platform components.

Processing Requirements and Permissions

(14) A method of processing specifications may define additional rules to determine if an expression value is legal for a property, beyond the restrictions imposed by the declared property type. The declared property type represents a minimum set of restrictions that must be enforced for every use of a property.

(15) If an associated expression or default value is not specified for a property, a method of processing specifications is permitted to reject that specification as erroneous. A method of processing specifications is permitted to construct a default expression, providing that default is made known to the developers. This decision may be made on a per property basis. If a property value is not required for a specific development activity, then the method of processing associated with this activity must accept a specification in which that property has no associated value.

(16) A method of processing specifications may impose additional restrictions on the use of property expressions whose value depends on the current mode of operation, or on bindings. For example, mode-dependent values may be allowed for some properties but disallowed for others. Mode-dependent property expressions may be disallowed entirely.

(17) A method of processing specifications may access and change field values of a record property value programmatically.

(18) A method of processing specifications may impose restrictions on the use of computed values in order to allow the computed value function to compute the value once and store the result as a cached value. For example, it may assume that the values of properties used in the computation have been declared through property associations. In that case the computed property value are not changed programmatically by a method of processing and can be determined at model instantiation time.

Examples

```
-- This is an AADL fragment inside a package
thread Producer
end Producer;

thread implementation Producer.Basic
properties
   Compute_Execution_Time  => 0ms..10ms in binding ( powerpc.speed_350Mhz );
   Compute_Execution_Time  => 0ms..8ms in binding ( powerpc.speed_450MHz );
end Producer.Basic ;

process Collect_Samples
end Collect_Samples;

system Software
end Software;

system implementation Software.Basic
subcomponents
   Sampler_A : process Collect_Samples;
   Sampler_B : process Collect_Samples
      {
          -- A property with a list of values
          Source_Text => ( "collect_samples.ads", "collect_samples.adb" ) ;
          Period => 50 ms;
      } ;
end Software.Basic;

device car
properties
  mine::Car_Length => 3.25 meter;
  mine::Position => ( x => 3, y => 4);
  mine::Car_Name => ( US => "Rabbit", Germany => "Golf" );
end car;

system Hardware
```

```
end Hardware;

system implementation Hardware.Basic
subcomponents
   Host_A: processor;
   Host_B: processor;
end Hardware.Basic ;

system Total_System
end Total_System;

system implementation Total_System.SW_HW
subcomponents
   SW : system Software.Basic;
   HW : system Hardware.Basic;
properties
   -- examples of contained property associations
   -- in a subcomponent of SW we are setting the binding to a
   -- component contained in HW
   Allowed_Processor_Binding => reference ( HW.Host_A )
                                applies to SW.Sampler_A;
   Allowed_Processor_Binding => reference ( HW.Host_B )
                                applies to SW.Sampler_B;
end Total_System.SW_HW;
```

12 Modes and Mode Transitions

(1) A *mode* represents an operational mode state, which manifests itself as a configuration of contained components, connections, and mode-specific property value associations. A configuration may be an execution platform configuration in the form of a set of processors, memories, buses, and devices; an application system configuration in the form of a set of threads communicating through connections or calls within or across processes and systems; or a source text operational mode within a thread, i.e., an execution behavior embedded in the source text itself. When multiple modes are declared for a component, a mode transition behavior declaration identifies which event and event data arrivals cause a mode switch and the new mode, i.e., a change to a different configuration. Exactly one mode is considered the current mode. The current mode determines the set of threads that are considered *active*, i.e., ready to respond to dispatches, and the connections that are available to transfer data and control. A set of modes may be inherited from the containing component.

(2) Mode transitions model dynamic operational behavior that represents switching between configurations and changes in components internal characteristics. Such transitions are initiated by mode transition triggering events, i.e., the arrival of events or event data on ports named in a mode transition or an event raised by the component itself (**self**). When declared for processes and systems, mode transitions model the switch between alternative configurations of active threads. When declared for execution platforms, mode transitions model the change between different execution platform configurations. When declared for threads and subprograms, modes may represent application logic that is encoded in the source text and may result in different associated property values for the thread or data component.

Syntax

```
modes_subclause ::=
    modes ( { mode | mode_transition }+ | none_statement )

requires_modes_subclause ::=
    requires modes ( { mode }+ | none_statement )

mode ::=
    defining_mode_identifier : [ initial ] mode
        [ { { mode_property_assocation }+ } ];

mode_transition ::=
    [ defining_mode_transition_identifier : ]
    source_mode_identifier
        -[ mode_transition_trigger { , mode_transition_trigger }* ]->
        destination_mode_identifier
       { { mode_transition_property_assocation }+ };

mode_transition_trigger ::=
    unique_port_identifier | self . event_source_identifier
    | processor . port_identifier

unique_port_identifier ::=
   [ subcomponent_identifier . ] port_identifier
```

```
in_modes ::=
    in modes ( mode_identifier { , mode_identifier }* )

component_in_modes ::=
    in modes ( ( ( mode_name { , mode_name }* ) | all ) )

mode_name ::= local_mode_identifier [ => subcomponent_mode_identifier ]

in_modes_and_transitions ::=
    in modes ( ( mode_or_transition { , mode_or_transition }* )

mode_or_transition ::=
        mode_identifier | mode_transition_identifier
```

Naming Rules

(N1) The defining mode and mode transition identifiers must be unique within the local namespace of the component classifier that contains the mode subclause. This means that modes declared in component types and in component implementations for the same type must have different identifiers.

(N2) The identifiers in a mode transition that refer to modes must exist in the local namespace of the component classifier that contains the mode subclause. In other words, mode transitions of a component can only refer to modes of the same component.

(N3) The unique port identifier must be either an **in** or **in out** event, data, or event data port identifier in the namespace of the associated component type or an **out** or **in out** event, data, or event data port in the namespace of the component type associated with the named subcomponent.

(N4) The mode and mode transition identifiers named in an **in modes** statement of a subcomponent or connection must refer to modes or mode transitions declared in the local namespace of the component implementation that contains the subcomponent or connection. The modes and mode transitions in the local name space are those declared in the `modes_subclause` or `requires_modes_subclause` of the component implementation or the component type.

(N5) The `subcomponent_mode_identifier` in a `component_in_modes` must refer to a required mode of the subcomponent for which the **in modes** is declared. If **all** is specified, then for each mode in the subcomponent there must be a mode with the same name in the containing component.

(N6) An **in modes** statement in a subcomponent or connection refinement declaration may be used to specify mode membership to replace the one, if any, in the declaration being refined. A connection refinement declaration without an **in modes** statement specifies that the connection is active in all modes.

Special rules apply for **in modes** in property associations (see Section 11.3).

(N7) For a contained property association with an **in modes** statement, the identifier must refer to modes or mode transitions of the last subcomponent named in the dot-separated identifier list of the **applies to** subclause, or to modes and mode transitions of the component containing the property association if the **applies to** subclause does not start with a subcomponent identifier.

Legality Rules

(L1) A mode or mode transition can be declared in any of the component categories.

(L2) If a component classifier contains mode declarations, one of those modes must be declared with the reserved word **initial**. If the component classifier extends another component classifier, the initial mode must have been declared in one of the ancestor component classifier. This rule does not apply to **requires modes** subclauses.

(L3) The set of transitions declared within a single component implementation must define a deterministic transition function. For each mode, there must exist exactly one transition, which can cause transition to another mode.

(L4) If a component has been declared with a `requires_mode_clause`, then a subcomponent declaration referencing this component classifier must include a `component_in_modes` that specifies a mapping for the inherited modes.

Standard Properties

```
Mode_Transition_Response: enumeration ( emergency, planned )
```

Semantics

Mode Sensitive Architecture

(3) The mode semantics described here focus on a single mode subclause. A system instance that represents the runtime architecture of an operational system can contain multiple components with their own mode transitions. The semantics of system-wide mode transitions are discussed in Section 13.6.

(4) A mode represents an operational state that is represented as a runtime configuration of contained components, connections, and mode-specific property value associations.

(5) A mode may represent a runtime configuration of systems, processes, thread groups and threads and their connections for a given operational state. In this case the modes are declared in thread groups, processes and systems, and **in modes** clauses indicate which subcomponents and connections are active in a given mode. In this case, only the threads that are part of the current mode are in the *suspended awaiting dispatch* state – responding to dispatch requests. All other threads are in the *suspended awaiting mode* state or *thread halted* state.

(6) A mode may also represent logical execution state within a thread. In this case the mode logic is embedded in the source text and may be represented as thread modes and mode-specific thread property values or mode-specific call sequences through the use of the **in modes** clause. Thread and subprogram internal modes effectively represent a set of behavioral states, whose state transition behavior can be specified through annex subclauses of the Behavior Annex notation (see Annex Document D).

(7) System and execution platform component declarations can have subcomponents with **in modes** clauses. In this case, only the execution platform components that are part of the current mode are accessible to software components. For example, only the processors and memories that are part of the current mode can be the target of bindings of application components active in that mode.

(8) For execution platform components a mode can represent an operational mode that is internal to the execution platform component. For example, a processor may execute at two execution speeds. The different execution speeds are reflected in mode-specific property values

(9) A component type or component implementation may contain several declared modes. Exactly one of those modes is the *current* mode. Initially, the initial mode is the current mode.

(10) A component that has modes itself can be a subcomponent of another component with modes. As a result, the component can be deactivated and activated as result of mode transitions in the enclosing component. On activation of a components with modes the `Resumption_Policy` property determines whether the initial mode is entered or the mode from the last deactivation is resumed.

(11) The **in modes** statement is declared as part of subcomponent declarations, subprogram call sequences, flow implementations, and property associations. It specifies the modes for which these declarations and property values hold. The mode identifiers refer to mode declarations in the modes subclause of the component classifier. If the **in modes** statement is not present, then the subcomponent, subprogram call sequence, flow implementation, or property association is part of all modes. If a property association has both mode-specific declarations and a declaration without an **in modes** statement, then the declaration without the **in modes** statement applies to those modes not covered by the mode-specific declarations.

(12) A component type may specify through a **requires modes** declaration that a subcomponent should inherit the modes of its containing component.. In this case the **in modes** declared as part of a subcomponent defines a mapping of the modes of the containing component to the inherited modes of the subcomponent. The modes of both may have the same names, or they may be different. Multiple modes of the containing component may be mapped to the same subcomponent mode.

(13) The **in modes** statement declared as part of connection declarations specify the modes or mode transitions for which these connection declarations hold. The mode identifiers refer to mode declarations in the modes subclause of the component implementation. If a connection is declared to be part of a mode transition, then the content of the ultimate source port is transferred to the ultimate destination port at the actual mode switch time. If the **in modes** statement contains only mode transitions, then the connection is part of the specified mode transitions, but not part of any particular mode. If the **in modes** statement is not present, then the connection is part of all modes.

(14) If the **in modes** statement is declared as part of a refinement, the newly named modes replace the modes specified in the declaration being refined.

Mode Transition

(15) The modes subclause declares a state machine describing the dynamic mode transition behavior between modes. The states of the state machine represent the different modes and the transitions specify the event(s) that can trigger a transition to the destination mode. Only one mode alternative represents the current mode at any one time.

(16) The arrival of an event or event data through an event or event data port that is named in one of the transitions is called a mode transition *trigger event*. A mode transition is triggered if one of the ports named in a mode transition out of the state representing the current mode causes the trigger event. If such an trigger event occurs and there is no transition out of the current mode naming the port with the trigger event, the trigger event is ignored with respect to mode transitions of the receiving modal component.

(17) A mode transition may actually be performed immediately if it is considered an *emergency*, or it may be performed in a *planned* fashion after currently executing threads complete their execution and the dispatches of a set of critical thread are aligned (see Section 13.6 for more detail). This is indicated by the `Mode_Transition_Response` property on the mode transition with the default being planned.

(18) If several trigger events occur logically simultaneously and affect different mode transitions out of the current mode, the order of arrival for the purpose of determining the mode transition is implementation dependent. If an `Urgency` property is associated with each port named in mode transitions, then the mode transition with the highest port urgency takes precedence. If several ports have the same urgency then the mode transition is chosen non-deterministically.

(19) Any change of the current mode has the effect of changing the property value in property associations with mode-specific values – as expressed by the **in modes** statement.

(20) A thread may execute different source text sequences under different modes declared within a thread. This may be represented by different compute execution times and different entrypoints or call sequences for different modes. The current mode at dispatch time determines the source text sequence to be executed. In other words, a thread-internal mode represents a behavioral state, in which the thread completes execution of one dispatch and awaits the next dispatch.

(21) The semantics of mode transitions between modes declared inside threads have been described in Section 5.4.5.

(22) Mode transitions within an execution platform component occur as a result of external or internal events, e.g., a processor may switch to a different execution speed. A mode transition within a thread or execution platform component does not affect the set of active threads, processors, devices, buses, or memories, nor does it affect the set of active connections external to the thread or execution platform component.

(23) A mode transition within a system, process, or thread group implementation has the effect of deactivating and activating threads to respond to dispatches, and changing the pattern of connections between components. Deactivated threads transition to the *suspended awaiting mode* state. Background threads that are not part of the new mode suspend performing their execution. Activated threads transition to the *suspended awaiting dispatch* state and start responding to dispatches. Previously suspended background threads that are part of the new mode resume performing execution once the transition into the new mode is complete. Threads that are part of both the old and new mode of a mode transition continue to respond to dispatches and perform execution. Ports that were connected in the old mode, may not be connected in the new mode and vice versa.

(24) Threads that are active in both the old and the new mode are dispatched in their usual manner; in the case of background threads, they continue in the *execute* state.

(25) At the time of the actual mode switch, any threads that were active in the old mode and are inactive in the new mode execute their `Deactivate_Entrypoint`.

(26) At the time of the actual mode switch, any threads that were inactive in the old mode and are active in the new mode execute their `Activate_Entrypoint`.

NOTES:

Mode transitions can only be triggered by the arrival of an event or event data on a named port or a **self** event. This means that trigger conditions based on the content of any data cannot be expressed in the mode transition declaration. Instead, they have to be connected to a thread whose source text interprets the data portion to identify the error type and then raise an appropriate event through an out event port that triggers the appropriate mode transition. Such a thread typically plays the role of a system health monitor that makes system reconfiguration decisions based on the nature and frequency of detected faults.

The default mode transition trigger conditions based on a set of ports is a disjunction. Other conditions can be modeled through the use of the Behavior Annex (see Annex Document D).

Processing Permissions and Requirements

(27) Every method for processing specifications must parse mode transition declarations and check the legality rules defined in this standard. However, a method of processing specifications need not define how to build a system from a specification that contains mode transition declarations. That is, complex behaviors that may have multiple modes of operation may be rejected by a method of building systems as an unsupported capability.

(28) In an actual distributed system, exact simultaneity among multiple events cannot be achieved. A system realization must use synchronization protocols sufficient to ensure that the causal ordering of event and data transfers defined by the logical temporal semantics of this standard are satisfied by the actual system, to the degree of assurance required by an application.

(29) A method of implementation is permitted to provide preservation of queue content for aperiodic and sporadic threads on a mode switch until the next activation. This is specified using the thread property `Active_Thread_Queue_Handling_Protocol`.

(30) A method of implementation is permitted to support a subset of the described protocols to handle threads that are in the performing computation state at the time instant of actual mode switch. They must document the chosen subset and its semantic behavior as part of the `Supported_Active_Thread_Handling_Protocol` property.

(31) A method of implementation may enforce that there is a mode-specific property value defined for each mode.

Examples

```
-- This is an AADL fragment inside a package
data Position_Type
end Position_Type;

process Gps_Sender
features
   Position: out data port Position_Type;
   -- if connected secondary position information is used to recalibrate
   SecondaryPosition: in data port Position_Type
                                 { Required_Connection => false;};
end Gps_Sender;

process implementation Gps_Sender.Basic
end Gps_Sender.Basic;

process implementation Gps_Sender.Secure
end Gps_Sender.Secure;

process GPS_Health_Monitor
features
   Backup_Stopped: out event port;
   Main_Stopped: out event port;
   All_Ok: out event port;
   Run_Secure: out event port;
   Run_Normal: out event port;
end GPS_Health_Monitor;

system Gps
features
   Position: out data port Position_Type;
   Init_Done: in event port;
end Gps;

system implementation Gps.Dual
subcomponents
   Main_Gps: process Gps_Sender.Basic in modes (Dualmode, Mainmode);
   Backup_Gps: process Gps_Sender.Basic in modes (Dualmode, Backupmode);
   Monitor: process GPS_Health_Monitor;
```

```
connections
   port Main_Gps.Position -> Position in modes (Dualmode, Mainmode);
   port Backup_Gps.Position -> Position in modes (Backupmode);
   port Backup_Gps.Position -> Main_Gps.SecondaryPosition
                                        in modes (Dualmode);
modes
   Initialize: initial mode;
   Dualmode : mode;
   Mainmode : mode;
   Backupmode: mode;
   Started: Initialize -[ Init_Done ]-> Dualmode;
   Dualmode -[ Monitor.Backup_Stopped ]-> Mainmode;
   Dualmode -[ Monitor.Main_Stopped ]-> Backupmode;
   Mainmode -[ Monitor.All_Ok ]-> Dualmode;
   Backupmode -[ Monitor.All_Ok ]-> Dualmode;
end Gps.Dual;

system implementation Gps.Secure extends Gps.Dual
subcomponents
   Secure_Gps: process Gps_Sender.Secure in modes ( Securemode );
connections
   port Secure_Gps.Position -> Position in modes ( Securemode );
modes
   Securemode: mode;
   SingleSecuremode: mode;
   Dualmode -[ Monitor.Run_Secure ]-> Securemode;
   Securemode -[ Monitor.Run_Normal ]-> Dualmode;
   Securemode -[ Monitor.Backup_Stopped ]-> SingleSecuremode;
   SingleSecuremode -[ Monitor.Run_Normal ]-> Mainmode;
   Securemode -[ Monitor.Main_Stopped ]-> Backupmode;
end Gps.Secure;
```

The following example illustrates inherited modes. A process declares a high fidelity and a low fidelity mode. All threads in the process respond to these two modes by performing high or low fidelity computation.

```
thread Calculate
features
  Incoming: in event data port;
  Outgoing: out event data port;
requires modes
  HighFidelity: initial mode;
  LowFidelity: mode;
```

```
end Calculate;
process Altitude
features
  doHigh: in event port;
  doLow: in event port;
  Incoming: in event data port;
  Outgoing: out event data port;
end Altitude;
process implementation Altitude.impl
subcomponents
  calc: thread Calculate in modes (LowFid => LowFidelity, HiFid => HighFidelity);
connections
  port Incoming -> calc.Incoming;
  port calc.Outgoing -> Outgoing;
modes
  HiFid: initial mode;
  LowFid:  mode;
  HiFid -[ doLow ]-> LowFid;
  LowFid -[ doHigh ]-> HiFid;
end Altitude.impl;
```

13 Operational System

(1) Component type and component implementation declarations are architecture design elements that define the structure and connectivity of a actual system architecture. They are component classifiers that must be instantiated to create a complete system instance. A complete system instance that represents the containment hierarchy of the actual system is created by instantiating a root system implementation and then recursively instantiating the subcomponents and their subcomponents. Once instantiated, a system instance can be completely bound, 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.

(2) A completely instantiated and bound system acts as a blueprint for a system build. Binary images are created and configured into load instructions according to the system instance specification in AADL.

13.1 System Instances

(1) A system instance represents the runtime architecture of a actual system that consists of application software components and execution platform components.

(2) A system instance is *completely instantiable* if the system implementation being instantiated is completely specified and completely resolved.

(3) A system instance is *completely instantiated and bound* if all threads are ultimately bound to a processor, all source text making up process address spaces are bound to memory, connections are bound to buses if their ultimate source and destinations are bound to different processors, and subprogram calls are bound to remote subprograms as necessary.

(4) A set of contained property associations can reflect property values that are specific to individual instances of components, ports, connections, provided and required access. These properties may represent the actual binding of components, as well as results of analysis, simulation, or actual execution of the completely instantiated and bound system. Thus, multiple sets of contained property associations can be associated with the same system instance to represent different system configurations.

Consistency Rules

(C1) A complete system instance must not contain incompletely specified subcomponents, ports, and subprograms. All processes in a completely instantiable system must contain at least one thread.

(C2) The `Required_Connection` property may be used to indicate that a port connection is required. In the case of the predeclared `Error` and `Complete` ports (see Section 5.4), connections are optional.

(C3) In a complete system instance, the required ports of all threads, devices, and processors must be the ultimate source or destination of semantic connections.

(C4) In a completely instantiable system, the subprogram calls of all threads must either be local calls or be bound to a remote subprogram whose thread is part of the same mode.

(C5) In a completely instantiable system, for every mode that is the source of mode transitions, there must be at least one mode transition that is the ultimate destination of a semantic connection whose ultimate source is part of the mode.

(C6) In a complete system instance, `aperiodic` and `sporadic` threads that are part of a given mode must have at least one connection to one of their **in** event ports or **in** event data ports.

(C7) For instantiable systems, all threads must be bindable to processors and all components representing source text must be bindable to memory.

(C8) The source text associated with all contained components of the system instance must be compliant with the specified component type, component implementation, and property associations.

(C9) In order to be considered complete, system instance must contain at least one thread, one processor and one memory component in its containment hierarchy to represent an application system that is executable on an execution platform, i.e., a processor with memory containing the application code and data.

(C10) If a system instance has processors of different types, then the execution time of threads must be specified for each processor type using the **in binding** statement, or a reference processor type must be identified through the `Reference_Processor` property.

Semantics

(5) A system instance represents an operational actual system. That actual system may be a stand-alone system or a system of systems. A system instance consists of application software and execution platform components. The component configuration, i.e., the hierarchical structure and interconnection topology of these components is statically known. The mode concept describes alternative statically known component configurations. The runtime behavior of the system allows for switching between these alternative configurations according to a mode transition specification.

(6) The actual system denoted by a system implementation can be built if the system is instantiable and if source text exists for all components whose properties refer to source text. This source text must be compliant with the AADL specification and the source text language semantics. Source text is compiled and linked to generate binary images. The binary images are loaded into memory and made accessible to threads in virtual address spaces of processes.

(7) In addition, there exists a kernel address space for every processor. This address space contains binary images of processor software and device driver software bound to the processor.

(8) Execution time of threads and subprograms is specified in units of time. If a system instance has processors of different types, then the execution time may vary from processor type to processor type. Execution time specific to a processor type can be specified using the **in binding** statement in a property association. Alternatively, the execution time can be specified with respect to a reference processor. The reference processor is specified through the `Reference_Processor` property. This property is declared as **inherit**, i.e., it can be declared at the system level and apply to all threads in a system. When a thread is bound to a specific processor type, its execution time, if not explicitly defined for this processor, will be adjusted according to a scaling factor between the reference processor and the target processor.

13.2 System Binding

(1) This section defines how binary images produced by compiling source units are assigned to and loaded onto processor and memory resources, taking into account requirements for component sharing and the interconnect topology between processors, memories and devices. The decisions and methods required to combine the components of a system to produce a actual system implementation are collectively called bindings.

Naming Rules

(N1) The `Allowed_Processor_Binding` property values must evaluate to a processor, virtual processor, or a system that contains a processor in its component containment hierarchy.

(N2) The `Allowed_Memory_Binding` property values must evaluate to a memory, a processor that contains memory or a system that contains a memory or a processor containing memory in its component containment hierarchy.

Legality Rules

(L1) The memory requirements of ports and data components are specified as property values of their data types or by properties on the port feature or data subcomponent. Those property associations can have binding-specific values.

Consistency Rules

(C1) Every mode-specific configuration of a system instance must have a binding of every process component to a (set of) memory component(s), and a binding of every thread component to a (set of) processor(s).

(C2) In the case of dynamic process loading, the actual binding may change at runtime. In the case of tightly coupled multi-processor configurations, such as dual core processors, the actual thread binding may change between members of an actual binding set of processors as these processors service a common set of thread ready queues.

(C3) Multiple software components may be bound to a single memory component.

(C4) A software component may be bound to multiple memory components.

(C5) A thread must be bound to a one or more processors. If it is bound to multiple processors, the processors share a ready queue, i.e., the thread executes on one processor at a time.

(C6) Multiple threads can be bound to a single processor.

(C7) All software components for a process must be bound to memory components that are all accessible from every processor to which any thread contained in the process is bound. That is, every thread is able to access every memory component into which the process containing that thread is loaded.

(C8) A shared data component must be bound to memory accessible by all processors to which the threads sharing the data component are bound.

(C9) For all threads in a process, all processors to which those threads are bound must have identical component types and component implementations. That is, all threads that are contained in a given process must all be executing on the same kind of processor, as indicated by the processor classifier reference value of the `Allowed_Processor_Binding_Class` property associated with the process. Furthermore, all those processors must be able to access the memory to which the process is bound.

(C10) The complete set of software components making up the functionality of the processor must be bound to memory that is accessible by that processor.

(C11) Each thread must be bound to a processor satisfying the `Allowed_Processor_Binding_Class` and `Allowed_Processor_Binding` property values of the thread.

(C12) The `Allowed_Processor_Binding` property may specify a single processor, thus specifying the exact processor binding. It may also specify a list of processor components or system components containing processor components, indicating that the thread is bindable to any of those processor components.

(C13) Each process must be bound to a memory satisfying the `Allowed_Memory_Binding_Class` and `Allowed_Memory_Binding` property values of the process. The `Allowed_Memory_Binding` property may specify a single memory component, thus specifying the exact memory binding. It may also specify a list of memory components or system components containing memory components, indicating that the process is bindable to any of those memory components.

(C14) The memory requirements of the binary images and the runtime memory requirements of threads and processes bound to a memory component must not exceed that memory's capacity. The execution time requirements of all threads bound to a processor must not exceed the schedulable cycles required to ensure that all thread timing requirements are met. These two constraints may be checked statically or dynamically. Runtime detection of such a memory capacity or timing requirements violation results in an error that the application system can choose to recover from.

Semantics

(2) A complete system instance is instantiated and bound by identifying the actual binding of all threads to processors, all binary images reflected in processes and other components to memory, and all connections to buses if they span multiple processors. The actual binding can be recorded for each component in the containment hierarchy by property associations declared within the system implementation.

(3) The actual binding must be determined within specified binding constraints. Binding constraints of application components to execution platform components are expressed by the allowed binding and allowed binding class properties for memory, processor, and bus. In the case of an allowed binding property, the execution platform component is identified by a sequence of '.' (dot) separated subcomponent names. This sequence starts with the subcomponent contained in the component implementation for which the property association is declared. Or the sequence begins with the subcomponent contained in the component implementation of the subcomponent or system implementation for which the property association is declared. This means that the property association representing the binding constraint or the actual binding may have to be declared as a component instance property association of a component that represents a common root of the components to be bound.

Processing Requirements and Permissions

(4) A method of building systems is permitted to require bindings of selected kinds to be fixed at development time, or to be fixed at the time of actual system construction. A method of building systems is permitted to allow bindings of selected kinds to change dynamically at runtime. For example, a method of building systems may require process to memory binding and loading to be fixed during actual system construction, and may require thread to processor bindings to be fixed at mode changes. Other choices are possible and permitted.

(5) A method of building systems must check and enforce the semantics and legality rules defined in this standard. Property associations may impose constraints on allowed bindings. The access semantics impose a number of constraints on allowed bindings for processes and threads to execution platform systems, and ultimately to processors and memories. In general, the semantic constraints depend on the particular software and hardware architecture interconnect topologies. In particular, for most hardware and operating system configurations all threads contained in a process must execute on the same processor. Such additional restrictions must be taken into account by the method of building systems. A method of building systems is otherwise permitted to make any partitioning and binding choices that are consistent with the semantics and legality rules of this standard.

NOTES:

If multiple processes share a component, then the physical memory to which the shared component is bound will appear in the virtual address space of all those processes. This physical memory is not necessarily addressed using either the same virtual address in different processes or the same physical address from different processors. An access property association may be used to specify different addresses used to access the same component from different processors.

The AADL supports binding-specific property values. This allows different property values to be specified for a component property whose values are sensitive to binding decisions.

Examples

```
package Deployment

system smp

end smp;


system implementation smp.s1

-- a multi-processor system

subcomponents

   p1: processor cpu.u1;

   p2: processor cpu.u1;
```

```
   p3: processor cpu.u1;
end smp.s1;

process p1
end p1;

process implementation p1.i1
subcomponents
   ta: thread t1.i1;
   tb: thread t1.i1;
end p1.i1;

thread t1
end t1;

thread implementation t1.i1
end t1.i1;

processor cpu
end cpu;

processor implementation cpu.u1
end cpu.u1;

system S
end S;

system implementation S.I
-- a system combining application components
-- with execution platform components
subcomponents
   p_a: process p1.i1;
   p_b: process p1.i1;
   up1: processor cpu.u1;
   up2: processor cpu.u1;
   ss1: system smp.s1;
properties
    Allowed_Processor_Binding => ( reference (up1), reference (up2) )
                                 applies to p_a.ta;
```

```
    Allowed_Processor_Binding => ( reference (up1), reference (up2) )
                                        applies to p_a.tb;

   -- ta is restricted to a subset of processors that tb can be bound to;
   -- since ta and tb are part of the same process they must be bound to the
   -- same processor in most hardware configurations
   Allowed_Processor_Binding => reference (ss1.p3 ) applies to p_b.ta;
   Allowed_Processor_Binding => reference (ss1 ) applies to p_b.tb;
end S.I;
end Deployment;
```

NOTES:

Binding properties are declared in the system implementation that contains in its containment hierarchy both the components to be bound and the execution platform components that are the target of the binding. Binding properties can also be declared separately for each instance of such a system implementation, either as part of the system instantiation or as part of a subcomponent declaration.

13.3 System Startup

(1) On system startup the system instance transitions from *system offline* to *system starting* (see Figure 22). **Start(system)** initiates the initialization of the execution platform components. In the case of processors, the binary images of the kernel address space are loaded into memory of each processor, and execution is started to initialize the execution platform software (see Figure 9). Loading into memory may take zero time, if the memory can be preloaded, e.g., PROM or flash memory.

(2) Once a processor is initialized (*Processor operational*), each processor initiates the initialization of virtual processors bound to the processor, if any (see Figure 9). When the virtual processors have completed their initialization they are operational (see Figure 10).

(3) When processors and virtual processors have entered their operational state (**started(processor)** and **started(vprocessor)**), the loading of the binary images of processes bound to the specific processor or its virtual processors into memory is initiated (see Figure 8). Process binary images are loaded in the memory component to which the process and its contained software components are bound. In a static process loading scenario, all binary images must be loaded before execution of the application system starts, i.e., thread initialization is initiated. In a dynamic process loading scenario, binary images of all the processes that contain a thread that is part of the current mode must be loaded.

(4) The maximum system initialization time can be determined as

- max(`Startup_Deadline`) of all processors and other hardware components
- + max(`Startup_Deadline`) of all virtual processors
- + max(`Load_Deadline`) of all processes
- + max(Process_Startup_Deadline) of all processes
- + max(`Initialize_Deadline`) of all threads.

(5) All software components for a process must be bound to memory components that are all accessible from every processor to which any thread contained in the process is bound. That is, every thread is able to access every memory component into which the binary image of the process containing that thread is loaded.

(6) Data components shared across processes must be bound to memory accessible by all processors to which the processes sharing the data component are bound.

(7) Thread initialization must be completed by the next hyperperiod of the initial mode. Once all threads are initialized, threads that are part of the initial mode enter the await dispatch state. If loaded, threads that are not part of the initial mode enter the suspend awaiting mode state (see Figure 5). At their first dispatch, the initial values of connected **out** or **in out** ports are made available to destination threads in their **in** or **in out** ports.

(8) This initialization model assumes independent initialization of threads, i.e., no ordering requirement (other than processes must be initialized first). If there is an ordering requirement the user can introduce an initialization mode, in which they can utilize the full power of AADL to specify thread execution dependencies.

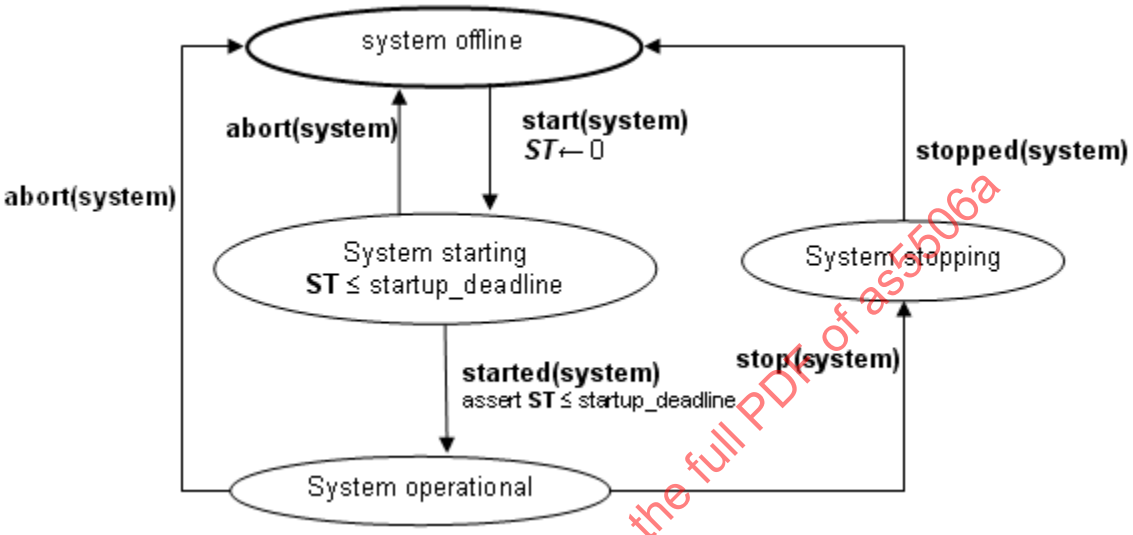


Figure 22 System Instance States, Transitions, and Actions

13.4 Normal System Operation

(1) Normal operation, i.e., the execution semantics of individual threads and transfer of data and control according to connection and shared access semantics, have been covered in previous sections. In this section we focus on the coordination of such execution semantics throughout a system instance.

(2) A system instance is called synchronized if all components use a globally synchronized reference time. A system instance is called asynchronous if different components use separate clocks with the potential for clock drift. The clock drift in asynchronous systems may be bounded, e.g., by resynchronizing the clocks.

(3) In a synchronized system, periodic threads are dispatched simultaneously with respect to a global clock. The hyperperiod of a set of periodic threads is defined to be the least common multiple of the periods of those threads.

(4) In a synchronized system, a raised event logically arrives simultaneously at the ultimate destination of all semantic connections whose ultimate source raised the event. In a synchronized system, two events are considered to be raised logically simultaneously if they occur within the granularity of the globally synchronized reference time. If several events are logically raised simultaneously and arrive at the same port or at different transitions out of the current mode in the same or different components, the order of arrival is implementation-dependent.

(5) In an asynchronous system the above hold within a synchronization domain, i.e., periodic threads are dispatched simultaneously and events arrive logically simultaneously. A method of implementing a system may provide coordination protocols for asynchronous system to provide simultaneity guarantees within a certain time granularity. Otherwise, the dispatches and event arrivals are considered independent across synchronization domains.

13.5 System Operation Modes

(1) The set of all modes specified for all components of a system instance form a set of concurrent mode state machines, whose composite state is called *system operation mode* (SOM). The set of possible SOMs is the cross product of the sets of modes for each component. That is, a SOM is a set of component modes, one mode for each component of the system. The initial SOM is the set of initial modes for each component.

(2) The set of possible SOMs can be reduced by taking into account mode combinations that are not reachable. For example, if a component instance with modes is itself not active under certain modes of one of its containing components, then these mode combinations can be excluded. Similarly, of a component inherits modes from a containing component, then only the mode combinations of the containing component need to be considered. Finally, reachability analysis of modes taking into account mode transition conditions may further reduce the set of SOMs to be considered.

(3) The discrete variable **Mode** denotes a SOM. That is, the variable **Mode** denotes a compound discrete state that is defined by the mode hybrid semantic diagrams for each component instance in the system. Note that the value of **Mode** will in general change at various instants of time during system operation, although not in a continuous time-varying way.

13.6 System Operation Mode Transitions

(1) A SOM transition is initiated whenever a mode transition trigger event (see also Section 12) occurs in any modal component in the system instance. A single sent event or event data can trigger a mode switch request in one or more components. In a synchronized system, this trigger event occurs logically simultaneously for all components and the resulting component mode transition requests are treated as a single SOM transition.

(2) Several events may occur logically simultaneously and may originate from different ports of the same component, e.g., through a `Send_Output` service call with multiple ports as parameter, or from different components at the same time. If they are semantically connected to transitions in different components that lead out of their current mode or to different transitions out of the same mode in one component, then events are treated as independent SOM transition initiations.

(3) If multiple SOM transition initiations occurs logically simultaneously, one is chosen in an implementation-dependent manner. If an `Urgency` property is associated with each port named in mode transitions, then the mode transition with the highest port urgency takes precedence. If several ports have the same urgency then the mode transition is chosen non-deterministically.

(4) A mode transition of a thread internal mode, i.e., a mode declared in the thread or one of its subprograms, that is triggered by the component itself or is triggered by an event or data arriving ata port of the thread, takes place at the next thread dispatch. For further detail see Section 5.4.5.

(5) A mode transition of a processor, device, bus, or memory internal mode, i.e., a mode declared in the component implementation, that is triggered by the component itself or is triggered by an external event, takes place immediately.

The next paragraphs address mode transitions that involve activation and deactivation of threads and connections.

(6) If a mode transition has a `Mode_Transition_Response` property value of `Emergency`, the `mode_transition_in_progress` state is entered in zero time and the actual SOM transition is performed immediately. In this case all threads in the performing state that have to be deactivated are aborted.

(7) After an event triggering a SOM transition request has arrived, the actual SOM transition occurs as a `Planned` response, if the mode transition does not have a `Mode_Transition_Response` property value of `Emergency`. In this case, execution continues under the old SOM until the dispatches of a critical set of periodic components (threads and devices), which are active in the old SOM, align at their hyperperiod. At that point the `mode_transition_in_progress` state is entered. A component is considered critical if it has a `Synchronized_Component` property value of true. If the set of critical components is empty, the `mode_transition_in_progress` state is entered immediately. This is indicated in Figure 23.

(8) by the guard of the function Hyper(**Mode**) on the transition from the `current_system_operation_mode` state to the `mode_transition_in_progress` state.

(9) Until the `mode_transition_in_progress` state is entered, any mode transition trigger event with the same or lower urgency that would result in a SOM transition from the current SOM is ignored. A trigger event with higher urgency supersedes the event under consideration for determining the SOM transition at the time the `mode_transition_in_progress` state is entered.

(10) A runtime transition between SOMs requires a non-zero interval of time, during which the system is said to be in transition between two system modes of operation. While a system is in transition, excluding the instants of time at the start and end of a transition, all mode transition trigger events are ignored with respect to mode transitions.

(11) The system is in a *mode transition in progress state* for a limited amount of time. This is determined by the maximum time required to perform all deactivations and activations of threads and connections (see below), increased to the next multiple of the hyperperiod of a non-zero set of critical components that continue to execute, i.e., periodic threads that are active in the old and in the new SOM and have a `Synchronized_Component` property value of true. This is shown in Figure 23.

(12) by the guard of the function Hyper(**Mode**)* on the transition from the `mode_transition_in_progress` state to the `current_system_operation_mode` state. After that period of time, the component is considered to operate in the new mode and the active threads in the new mode start to execute.

(13) At the time the `mode_transition_in_progress` state is entered there are several kinds of threads

- *Continuing threads*: threads that continue to execute because they are active in the old mode and active in the new mode;
- *Activated threads*: threads that are inactive in the old mode and are active in the new mode;
- *Deactivated threads*: threads that are active in the old mode, not active in the new mode, and whose dispatches align at the hyperperiod;
- *Zombie threads*: aperiodic, sporadic, timed, or hybrid threads as well as periodic threads that are active in the old mode and not active in the new mode and whose dispatches are not aligned at the hyperperiod, i.e., they may still be in the perform thread computation state (see Figure 5) and may have events or event data in their port queues.

(14) At the instant of time the `mode_transition_in_progress` state is entered, connections that are part of the old SOM and not part of the new SOM are disabled.

(15) While in the `mode_transition_in_progress` state, for all connections between data ports that are declared to be active in the current mode transition and whose source threads have completed execution at the time the `mode_transition_in_progress` state is entered, data is transferred from the **out** data port such that its value becomes available at the first dispatch of the receiving thread.

(16) At the instant in time the `mode_transition_in_progress` state is exited, connections that are not part of the old SOM and are part of the new SOM are enabled. At that time the new current SOM is entered and the following hold for data port connections. The data value of the **out** data port of the source thread is transferred into the **in** data port variable of the newly enabled thread, unless the **in** data port of the destination thread is the destination of a connection active in the current mode transition.

(17) While in the `mode_transition_in_progress` state, all continuing threads continue to get dispatched and execute, but will only use connections that are active in both the old and the new SOM.

(18) When the `mode_transition_in_progress` state is entered, *thread* **exit(Mode)** is triggered to deactivate all threads that are part of the old mode and not part of the new mode, i.e., the set of deactivated threads. This results in the execution of deactivation entrypoints for those threads (see Figure 5).

(19) When the `mode_transition_in_progress` state is entered, *thread* **enter(Mode)** is triggered to activate threads that are part of the new mode and not part of the old mode, i.e., the set of activated threads. This permits those threads to execute their activation entrypoints (see Figure 5).

(20) There is no requirement that all deactivation entrypoint executions complete before any activation entrypoint executions start. The maximum execution time for the deactivations and activations is the maximum deadline of the respective entrypoints.

(21) Zombie threads may be stopped at actual mode switch time, they may be suspended, or they may complete their execution while mode transition is in progress. The `Active_Thread_Handling_Protocol` property specifies for each such thread what action is to be taken at the time the `mode_transition_in_progress` state is entered.

(22) The default action is to `stop` the execution of the zombie thread and execute its recover entrypoint indicating an stop for deactivation - effectively handling them like deactivated threads. This permits the thread to recover to a consistent state for future activation, including the release of resources held by the thread. Upon completion of the recover entrypoint, execution the thread enters the suspended awaiting mode state; event and event data port queues of the thread are flushed by default or remain in the queue until the thread is activated again as specified by the `Active_Thread_Queue_Handling_Protocol` property. If the thread was executing a remotely called subprogram, the current dispatch execution of the calling thread of a call in progress or queued call is also aborted. In this case the recover entrypoint deadline is taken into account when determining the duration of the `mode_transition_in_progress` state.

(23) Other actions are project-specific implementor provided action and may include:

- `Suspend` the execution of the zombie thread for resumption the next time the thread is part of a new mode. This action is only safe to use if the thread does not hold the lock to a shared resource.
- `Complete_one` permits the thread to complete the execution of its current dispatch and invoke its deactivation entrypoint. Any remaining queued events, or event data may be flushed, or remain in the queue until the thread is activated again, as specified by the `Active_Thread_Queue_Handling_Protocol` property. The output is held and communicated according to the connections when the thread is reactivated. In this case the execution of the zombie thread competes for execution platform resources.
- `Complete_all` permits the thread to finish processing all events or event data in its queues at the time the actual mode switch state is entered. The output is held and communicated according to the connections when the thread is reactivated. When execution completes the deactivation entrypoint is executed. In this case the execution of the zombie thread competes for execution platform resources.

(24) At the next multiple of the SOM transition hyperperiod the system instance enters `current_system_operation_mode` state and starts responding to new requests for SOM transition.

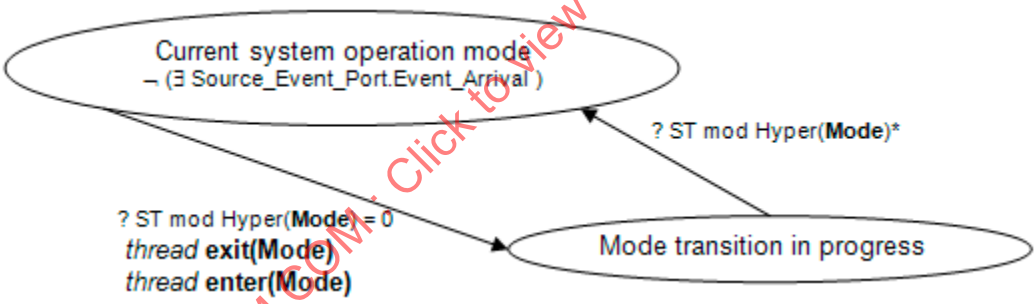


Figure 23 System Mode Transition Semantics

(25) The synchronization scope for **enter(Mode)** consists of all threads that are contained in the system instance that were inactive and are about to become active. The synchronization scope for **exit(Mode)** contains all threads that are contained in the system instance that were active and are to become inactive. The edge labels **enter(Mode)** and **exit(Mode)** also appear in the set of concurrent semantic automata derived from the mode declarations in a specification. That is, **enter(Mode)** and **exit(Mode)** transitions for threads occur synchronously with a transition from the `current_system_operation_mode` state to the `mode_transition_in_progress` state.

(26) The description of this time-coordinated transitioning of system operation mode assumes a synchronous system, i.e., a single synchronization domain. In the case of an asynchronous system, multiple synchronization domains exist in a system. In this case, the coordinated activation and deactivation of threads and connections as part of a system operation mode transition must be ensured within each synchronization domain. In the case of an asynchronous system, coordination protocols may be supported to coordinate system operation mode transition across synchronization domains within bounded time.

13.7 System-wide Fault Handling, Shutdown, and Restart

(1) Thread unrecoverable errors result in transmission of event data on the Error port of the appropriate thread, processor, or device. The ultimate destination of this semantic connection can be a thread or set of threads whose role is that of a system health monitor and system configuration manager. Such threads make decisions about appropriate fault handling actions to take. Such actions include raising of events to trigger mode switches, e.g., to request SOM transitions.

Runtime Services

(2) Several runtime services are provided to control starting and stopping of the system, processors, virtual processors, and processes. They may be explicitly called by application source code, or they may be called by an AADL runtime system that is generated from an AADL model.

(3) `Current_System_Mode` returns a value corresponding to the active system operation mode (SOM).

```
subprogram Current_System_Mode
features
  ModeID: out parameter <implementor-specific>; -- ID of the mode
end Current_System_Mode;
```

(4) `Set_System_Mode` indicates the next SOM the runtime system must switch to. System modes are assumed to have identifying numbers.

```
subprogram Set_System_Mode
features
  ModeID: in parameter <implementor-specific>; -- ID of the SOM
end Set_System_Mode;
```

(5) The following subprograms take a process, virtual processor, or processor ID as parameter. The ID representation is determined by the runtime system.

(6) `Stop_Process` is called for a controlled shut-down of a process, i.e., all of its contained threads, whether executing, awaiting a dispatch, or not part of the current mode, are given a chance to execute their finalize entrypoint before being halted.

(7) `Abort_Process` is called for a shut-down of a process due to an anomaly. All of its contained threads are terminated immediately.

(8) `Stop_Virtual_Processor` is called to initiate a transition to the virtual processor stopping state at the next hyperperiod. This has the effect of initiating `Stop_Virtual_Processor` for all virtual processors bound to the virtual processor, and `Stop_Process` for all processes bound to the virtual processor.

(9) `Abort_Virtual_Processor` is called for a shut-down of a virtual processor due to an anomaly. All virtual processors and processes bound to the virtual processor are aborted.

(10) `Stop_Processor` is called to initiate a transition to the processor stopping state at the next hyperperiod. This has the effect of initiating `Stop_Virtual_Processor` for all virtual processors bound to the processor, and `Stop_Process` for all processes bound to the processor.

(11) `Abort_Processor` is called for a shut-down of a processor due to an anomaly. All virtual processors and processes bound to the processor are aborted.

(12) `Stop_System` is called to initiate a transition to the system stopping state, which will initiate a Stop_Processor for all processors in the system.

(13) `Abort_System` is called for a shut-down of the system due to an anomaly. All processors in the system are aborted.

Processing Requirements and Permissions

(14) This standard does not require that source text be associated with a software or execution platform category. However, a method of implementing systems may impose this requirement as a precondition for constructing a actual system from a specification.

(15) A system instance represents the runtime architecture of an application system that is to be analyzed and processed. A system instance is identified to a tool by a component classifier reference to an instantiable system implementation. For example, a tool may allow a system classifier reference to be supplied as a command line parameter. Any such externally identified component specification must satisfy all the rules defined in this specification for system instances.

(16) A method of building systems is permitted to only support static process loading.

(17) A method of building systems is permitted to create any set of loadable binary images that satisfy the semantics and legality rules of this standard. For example, a single load image may be created for each processor that contains all processes and threads executed by that processor and all source text associated with devices and buses accessible by that processor. Or a separate load image may be created for each process to be loaded into memory to make up the process virtual address space, in addition to the kernel address space created for each processor.

(18) A process may define a source namespace for the purpose of compiling source programs, define a virtual address space, and define a binary image for the purpose of loading. A method of building systems is permitted to separate these functions. For example, processes may be compiled and pre-linked as separate programs, followed by a secondary linking to combine the process binary images to form a load image.

(19) A method of building systems is permitted to compile, link and load a process as a single source program. That is, a method of building systems is permitted to impose the additional requirement that all associated source text for all threads contained in a process form a legal program as defined in the applicable programming language standard.

(20) If two software components that are compiled and linked within the same namespace have identical component types and implementations, or the intersection of their associated source text compilation units is non-empty, then this must be detected and reported.

(21) A method of building systems is permitted to omit loading of processor, device driver, and bus protocol software in a processor kernel address space if none of the threads bound to that processor need to access or execute that software.

(22) This standard supports static virtual memory management, i.e., permits the construction of systems in which binary images of processes are loaded during system initialization, before a system begins operation.

(23) Also permitted are methods of dynamic virtual memory management or dynamic library linking after process loading has completed and thread execution has started. However, any method for implementing a system must assure that all deadline properties will be satisfied for each thread.

(24) An alternative implementation of the process and thread state transition sequences is permitted in which a process is loaded and initialized each time the system changes to a mode of operation in which any of the containing threads in that process are active. This process load and initialize replaces the perform thread activate action in the thread state transition sequence as well as the process load action in the process state transition sequence. These alternative semantics may be adopted for any designated subset of the processes in a system. All threads contained in a process must obey the same thread semantics.

14 Layered System Architectures

(1) Layering of system architectures is supported in AADL in several ways:

- Hierarchical containment of components;
- Layered use of threads for processing and services;
- Layered virtual machine abstraction of the execution platform.

(2) The hierarchical containment of components is the result of modeling a system as a component architecture and decomposing its capabilities hierarchically. This is supported by the abstract component category as well as the software components, the hardware components, and the system components. Interaction between components at different levels of the system hierarchy is managed and controlled by the component features. Connections between the components must follow the component hierarchy.

(3) Threads and devices represent active components in a system. Threads and devices interact with other threads and devices through directional flow of events and data, representing a pipeline architecture pattern. Threads also may call on services of other threads through subprogram service calls. These thread services often are organized into service layers in that threads of one layer call on services of the same or lower layer, but not higher layers. Multi-tiered e-business architectures are examples of this architecture pattern. Threads and related components of different service layer can be organized into separate packages and access to packages can be restricted by convention to follow the layering hierarchy. Packages or components such as threads can be attributed with a `Service_Layer` property that can be used by a design rule checker to enforce such layering.

(4) Execution platform components virtual processor and virtual bus can represent virtual machine abstractions. For example, a virtual processor may represent a language runtime system such as the Ada runtime system, which is responsible for scheduling the execution of Ada tasks. This runtime system may be implemented on top of another virtual processor, namely a real-time operating system. Virtual processors and processors provide the runtime service calls described in *Runtime Support* in Section 5.4 and Section 8.3, as well as services declared explicitly as part of the features of a (virtual) processor type. Similarly, a virtual bus may represent a protocol that is implemented in terms of a lower level protocol.

(5) AADL can be used to represent this system layering as follows: The implementation of an execution platform component can be described by system components. For example, a processor that represents hardware and an operating system can be described by an application software system to represent the operating system and a lower level execution platform system that includes a lower-level processor abstraction. Similarly, the internal implementation of a device, e.g., a digital camera, can be described as a system that consists of image processing and transfer software as well as processors, internal memory, and USB bus interface, and CCD sensors as devices. A map data base may be a highly abstracted memory component type, whose implementation consists of database software and data components that represent the database content together with physical disks as lower-level memory components and a processor that acts as dedicated database server.

(6) System implementations can be associated with their respective processor, virtual processor, memory, bus, virtual bus, and device types and implementations through the `Implemented_As` property, which takes classifiers as its value.

(7) In the case of processors, virtual processors, and devices, these system implementations can be used as plug-replaceable patterns for the original platform components, when an instance model is created. The pattern is called plug-replaceable because the system component implementing the execution platform element implements the interface defined by the execution platform element type.

(8) In the case of a virtual bus or bus a connection bound to a bus or virtual bus is interpreted during model instantiation as rerouting the connection from its source to a source feature in the system implementation of the bus or virtual bus, and a second connection from a predeclared destination feature to the destination of the original connection. For example, the system implementation realizing a virtual bus may consist of a sender thread with a predeclared source port and a receiver thread with a predeclared destination port. Alternatively, the connection can be mapped into `Send_Output` and `Receive_Input` service calls (see *Runtime Support* in Section 8.3) to reflect an API-based functional service architecture.

(9) In the case of a memory component, read and write accesses to data components that are bound to a memory component can be mapped into read and write accesses to data components in the system implementation of the memory abstraction, or interpreted as retrieve and store service calls to the subprogram access features of the system component.

15 Lexical Elements

(1) The text of an AADL description consists of a sequence of lexical elements, each composed of characters. The rules of composition are given in this section.

15.1 Character Set

(1) The only characters allowed outside of comments are the graphic_characters and format_effectors.

Syntax

```
character ::= graphic_character | format_effector
            | other_control_character

graphic_character ::= identifier_letter | digit | space_character
                    | special_character
```

Semantics

(2) The character repertoire for the text of an AADL specification consists of the collection of characters called the Basic Multilingual Plane (BMP) of the ISO 10646 Universal Multiple-Octet Coded Character Set, plus a set of format_effectors and, in comments only, a set of other_control_functions; the coded representation for these characters is implementation defined (it need not be a representation defined within ISO-10646-1).

(3) The description of the language definition in this standard uses the graphic symbols defined for Row 00: Basic Latin and Row 00: Latin-1 Supplement of the ISO 10646 BMP; these correspond to the graphic symbols of ISO 8859-1 (Latin-1); no graphic symbols are used in this standard for characters outside of Row 00 of the BMP. The actual set of graphic symbols used by an implementation for the visual representation of the text of an AADL specification is not specified.

(4) The categories of characters are defined as follows:

identifier_letter

upper_case_identifier_letter | lower_case_identifier_letter

upper_case_identifier_letter

Any character of Row 00 of ISO 10646 BMP whose name begins "Latin Capital Letter".

lower_case_identifier_letter

Any character of Row 00 of ISO 10646 BMP whose name begins "Latin Small Letter".

digit

One of the characters 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9.

space_character

The character of ISO 10646 BMP named "Space".

special_character

Any character of the ISO 10646 BMP that is not reserved for a control function, and is not the space_character, an identifier_letter, or a digit.

format_effector

The control functions of ISO 6429 called character tabulation (HT), line tabulation (VT), carriage return (CR), line feed (LF), and form feed (FF).

other_control_character

Any control character, other than a format_effector, that is allowed in a comment; the set of other_control_functions allowed in comments is implementation defined.

(5) The following names are used when referring to certain special_characters:

Symbol	Name	Symbol	Name
"	quotation mark	:	colon
#	number sign	;	semicolon
=	equals sign	(	left parenthesis
)	Right parenthesis	_	underline
+	plus sign	[	left square bracket
,	Comma	]	right square bracket
-	Minus	{	left curly bracket
.	Dot	}	right curly bracket

Implementation Permissions

(6) In a nonstandard mode, the implementation may support a different character repertoire; in particular, the set of symbols that are considered identifier_letters can be extended or changed to conform to local conventions.

NOTES:

Every code position of ISO 10646 BMP that is not reserved for a control function is defined to be a graphic_character by this standard. This includes all code positions other than 0000 - 001F, 007F - 009F, and FFFE - FFFF.

15.2 Lexical Elements, Separators, and Delimiters

Semantics

(1) The text of an AADL specification consists of a sequence of separate *lexical elements*. Each lexical element is formed from a sequence of characters, and is either a delimiter, an identifier, a reserved word, a numeric_literal, a character_literal, a string_literal, or a comment. The meaning of an AADL specification depends only on the particular sequences of lexical elements that form its compilations, excluding comments.

(2) The text of an AADL specification is divided into lines. In general, the representation for an end of line is implementation defined. However, a sequence of one or more format_effectors other than character tabulation (HT) signifies at least one end of line.

(3) In some cases an explicit *separator* is required to separate adjacent lexical elements. A separator is any of a space character, a format_effector, or the end of a line, as follows:

- A space character is a separator except within a comment, a string_literal, or a character_literal.
- Character tabulation (HT) is a separator except within a comment.
- The end of a line is always a separator.

(4) One or more separators are allowed between any two adjacent lexical elements, before the first, or after the last. At least one separator is required between an identifier, a reserved word, or a numeric_literal and an adjacent identifier, reserved word, or numeric_literal.

(5) A *delimiter* is either one of the following special characters

( ) [ ] { } , . : ; = * + -

(6) or one of the following *compound delimiters* each composed of two or three adjacent special characters

:: => +=> -> <-> .. –[ ]–> { **}**

(7) Each of the special characters listed for single character delimiters is a single delimiter except if that character is used as a character of a compound delimiter, or as a character of a comment, string_literal, character_literal, or numeric_literal.

(8) The following names are used when referring to compound delimiters:

Delimiter	Name
::	qualified name separator
=>	association
+=>	additive association
->	directional connection
<->	bidirectional connection
..	interval
-[	left step bracket
]->	right step bracket
{**	begin annex
**}	end annex

Processing Requirements and Permissions

(9) An implementation shall support lines of at least 200 characters in length, not counting any characters used to signify the end of a line. An implementation shall support lexical elements of at least 200 characters in length. The maximum supported line length and lexical element length are implementation defined.

15.3 Identifiers

(1) Identifiers are used as names. Identifiers are case insensitive.

Syntax

```
identifier ::= identifier_letter {[underline] letter_or_digit}*
letter_or_digit ::= identifier_letter | digit
```

- An identifier shall not be a reserved word.
- For the lexical rules of identifiers, the rule of whitespace as token separator does not apply. In other words, identifiers do not contain spaces or other whitespace characters.

Legality Rules

(L1) An identifier must be distinct from the reserved words of the AADL.

Semantics

(2) All characters of an identifier are significant, including any underline character. Identifiers differing only in the use of corresponding upper and lower case letters are considered the same.

Processing Requirements and Permissions

(3) In a nonstandard mode, an implementation may support other upper/lower case equivalence rules for identifiers, to accommodate local conventions.

(4) In non-standard mode, a method of implementation may accept identifier syntax of any programming language that can be used for software component source text.

Examples

```
Count          X          Get_Symbol   Ethelyn           Garçon
Snobol_4       X1         Page_Count   Store_Next_Item   Verrűckt
```

15.4 Numerical Literals

(1) There are two kinds of numeric literals, real and integer. A real_literal is a numeric_literal that includes a point; an integer_literal is a numeric_literal without a point.

Syntax

```
numeric_literal ::= integer_literal | real_literal
integer_literal ::= decimal_integer_literal | based_integer_literal
real_literal ::= decimal_real_literal
```

15.4.1 Decimal Literals

(1) A decimal literal is a numeric_literal in the conventional decimal notation (that is, the base is ten).

Syntax

```
decimal_integer_literal ::= numeral [ positive_exponent ]
decimal_real_literal ::= numeral . numeral [ exponent ]
numeral ::= digit {[underline] digit}*
exponent ::= E [+] numeral | E - numeral
positive_exponent ::= E [+] numeral
```

Semantics

(2) An underline character in a numeral does not affect its meaning. The letter E of an exponent can be written either in lower case or in upper case, with the same meaning.

(3) An exponent indicates the power of ten by which the value of the decimal literal without the exponent is to be multiplied to obtain the value of the decimal literal with the exponent.

Examples

```
12        0         1E6       123_456    -- integer literals
12.0      0.0       0.456          3.14159_26     -- real literals
```

15.4.2 Based Literals

(1) A based literal is a numeric_literal expressed in a form that specifies the base explicitly.

Syntax

```
based_integer_literal ::= base # based_numeral # [ positive_exponent ]
base ::= digit [ digit ]
based_numeral ::= extended_digit {[underline] extended_digit}
extended_digit ::= digit | A | B | C | D | E | F | a | b | c | d | e | f
```

Legality Rules

(L1) The base (the numeric value of the decimal numeral preceding the first #) shall be at least two and at most sixteen. The extended_digits A through F represent the digits ten through fifteen respectively. The value of each extended_digit of a based_literal shall be less than the base.

Semantics

(2) The conventional meaning of based notation is assumed. An exponent indicates the power of the base by which the value of the based literal without the exponent is to be multiplied to obtain the value of the based literal with the exponent. The base and the exponent, if any, are in decimal notation.

(3) The extended_digits A through F can be written either in lower case or in upper case, with the same meaning.

Examples

```
2#1111_1111#    16#FF#      016#0ff#    -- integer literals of value 255
2#1110_0000#    16#E#E1     8#340#      -- integer literals of value 224
```

15.5 String Literals

(1) A string_literal is formed by a sequence of graphic characters (possibly none) enclosed between two quotation marks used as string brackets.

Syntax

```
string_literal ::= "{string_element}*"
string_element ::= "" | non_quotation_mark_graphic_character
```

(2) A string_element is either a pair of quotation marks (""), or a single graphic_character other than a quotation mark.

Semantics

(3) The sequence of characters of a string_literal is formed from the sequence of string_elements between the bracketing quotation marks, in the given order, with a string_element that is "" becoming a single quotation mark in the sequence of characters, and any other string_element being reproduced in the sequence.

(4) A null string literal is a string_literal with no string_elements between the quotation marks.

NOTES:

An end of line cannot appear in a string_literal.

Examples

```
"Message of the day:"
""                              --  a null string literal
" "   "A"   """"                --  three string literals of length 1
"Characters such as $, %, and } are allowed in string literals"
```

15.6 Comments

(1) A comment starts with two adjacent hyphens and extends up to the end of the line.

Syntax

```
comment ::= --{non_end_of_line_character}*
```

- A comment may appear on any line of a program.

Semantics

(2) The presence or absence of comments has no influence on whether a program is legal or illegal. Furthermore, comments do not influence the meaning of a program; their sole purpose is the enlightenment of the human reader.

Examples

```
--  this is a comment

end;  --  processing of Line is complete

--  a long comment may be split onto
--  two or more consecutive lines

----------------  the first two hyphens start the comment
```


15.7 Reserved Words

(1) The following are the AADL reserved words. Reserved words are case insensitive.

aadlboolean	**aadlinteger**	**aadlreal**	**aadlstring**
abstract	**access**	**all**	**and**
annex	**applies**	**binding**	**bus**
calls	**classifier**	**compute**	**connections**
constant	**data**	**delta**	**device**
end	**enumeration**	**event**	**extends**
false	**feature**	**features**	**flow**
flows	**group**	**implementation**	**in**

inherit	**initial**	**inverse**	**is**
list	**memory**	**mode**	**modes**
none	**not**	**of**	**or**
out	**package**	**parameter**	**path**
port	**private**	**process**	**processor**
properties	**property**	**prototypes**	**provides**
public	**range**	**record**	**reference**
refined	**renames**	**requires**	**self**
set	**sink**	**source**	**subcomponents**
subprogram	**system**	**thread**	**to**
true	**type**	**units**	**virtual**
with			

NOTES:

The reserved words appear in lower case boldface in this standard. Lower case boldface is also used for a reserved word in a string_literal used as an operator_symbol. This is merely a convention – AADL specifications may be written in whatever typeface is desired and available.

A change bar (|) located in the left margin is for the convenience of the user in locating areas where technical revisions, not editorial changes, have been made to the previous issue of this document. An (R) symbol to the left of the document title indicates a complete revision of the document, including technical revisions. Change bars and (R) are not used in original publications, nor in documents that contain editorial changes only.

Predeclared Property Sets

Normative

(1) The set of predeclared property sets `Deployment_Properties`, `Thread_Properties`, `Timing_Properties`, `Memory_Properties`, `Programming_Properties`, and `Modeling_Properties` is part of every AADL specification. It defines properties for AADL model elements that are defined in the core of the AADL. These property sets may not be modified by the modeler. `Deployment_Properties` contains properties related to the deployment of the embedded application on the execution platform. `Thread_Properties` contains properties that characterize threads and their features. `Timing_Properties` contains properties related to execution timing. `Memory_Properties` contains properties related to memory as storage, data access, and device access. `Programming_Properties` contains properties for relating AADL models to application programs. `Modeling_Properties` contains properties related to the AADL model itself.

(2) The property set `AADL_Project` is a part of every AADL specification. It defines property enumeration types and property constants that can be tailored for different AADL projects and site installations. These definitions allow for tailoring of the predeclared properties through changes to these predeclared property types and property constants.

(3) The property types, property definitions, and property constants of these predeclared property sets can be named with or without property set name qualification.

NOTES:

All predeclared properties and user-defined properties are applicable to components of the category **abstract** without listing this category in the **applies to** statement of the property definition.

NOTES:

In accordance with the naming rules for references to items defined in the predeclared property sets, the declarations in these property set refer to enumeration types and property constants declared in the `AADL_Project` property set without a qualifying property set name.

A.1 Predeclared Deployment Properties

(1) The properties of the predeclared property set named `Deployment_Properties` record binding constraints and actual bindings of software components to hardware components, i.e., of threads and virtual processors to virtual processors and processors, of processes and data components to memory, and of connections to virtual buses and buses, and of virtual buses to virtual buses, buses, virtual processors, and processors.

Property set Deployment_Properties **is**

```
Allowed_Processor_Binding_Class:
   inherit list of classifier (processor, virtual processor, system)

   applies to (thread, thread group, process, system, virtual processor,
device);
```

The `Allowed_Processor_Binding_Class` property specifies a set of virtual processor, processor and system classifiers. These component classifiers constrain the set of candidate virtual processors and processors for binding to the subset that satisfies the component classifier.

The value may be inherited from the containing component.

If this property has no associated value, then all processors specified in the `Allowed_Processor_Binding` are acceptable candidates.

Allowed_Processor_Binding: **inherit list of reference** (processor, virtual processor, system)

applies to (thread, thread group, process, system, virtual processor, device);

The Allowed_Processor_Binding property specifies the set of virtual processors and processors that are available for binding. The set is specified by a list of virtual processor, processor and system component names. System names represent the processors contained in them.

If the property is specified for a thread, the thread can be bound to any of the specified set of virtual processors or processors for execution. If the property is specified for a thread group, process, or system, then it applies to all contained threads, i.e., the contained threads inherit the property association unless overridden. If this property is specified for a device, then the thread associated with the device driver code can be bound to one of the set of processors for execution. The Allowed_Processor_Binding property may specify a single processor, thus specifying the exact processor binding.

The allowed binding may be further constrained by the processor classifier reference specified in the Allowed_Processor_Binding_Class property.

If this property has no associated value, then all processors declared in n AADL specification are acceptable candidates.

Actual_Processor_Binding: **inherit list of reference** (processor, virtual processor)

applies to (thread, thread group, process, system, virtual processor, device);

A thread is bound to the processor specified by the Actual_Processor_Binding property. The process of binding threads to processors determines the value of this property. If there is more than one processor listed, a scheduler will dynamically assign the thread to one at a time. This allows modeling of multi-core processors without explicit binding to one of the cores.
If a device is bound to a processor this indicates the binding of the device driver software.
A virtual processor may be bound to a processor. This indicates that the virtual processor executes on the processor it is bound to.
Threads, devices, and virtual processors can be bound to virtual processors, which in turn are bound to virtual processors or processors.

Allowed_Memory_Binding_Class:
inherit list of classifier (memory, system, processor)

applies to (thread, thread group, process, system, device, data, data port, event data port, subprogram, processor);

The Allowed_Memory_Binding_Class property specifies a set of memory, device, and system classifiers. These classifiers constrain the set of memory components in the Allowed_Memory_Binding property to the subset that satisfies the component classifier.

The value of the Allowed_Memory_Binding property may be inherited from the component that contains the component or feature.

If this property has no associated value, then all memory components specified in the Allowed_Memory_Binding are acceptable candidates.

Allowed_Memory_Binding: **inherit list of reference** (memory, system, processor)

applies to (thread, thread group, process, system, device, data, data port, event data port, subprogram, processor);

Code and data produced from source text can be bound to the set of memory components that is specified by the Allowed_Memory_Binding property. The set is specified by a list of memory

and system component names. System names represent the memories contained in them. The Allowed_Memory_Binding property may specify a single memory, thus specifying the exact memory binding.

The allowed binding may be further constrained by the memory classifier specified in the Allowed_Memory_Binding_Class.

The value of the Allowed_Memory_Binding property may be inherited from the component that contains the component or feature.

If this property has no associated value, then all memory components declared in an AADL specification are acceptable candidates.

Actual_Memory_Binding: **inherit list of reference** (memory)

applies to (thread, thread group, process, system, processor, device, data, data port, event data port, subprogram);

Code and data from source text is bound to the memory specified by the Actual_Memory_Binding property.

Allowed_Connection_Binding_Class:
inherit list of classifier(processor, virtual processor, bus, virtual bus, device, memory)

applies to (feature, connection, thread, thread group, process, system, virtual bus);

The Allowed_Connection_Binding_Class property specifies a set of execution platform classifiers to constrain the binding of connections and virtual buses. The named component classifiers must belong to a processor, virtual processor, bus, virtual bus, device, or memory category, i.e., any execution platform component that supports communication between threads. When specified for a feature it indicates a binding constraint for all connections through that feature, e.g., any protocol assumptions a component makes about its communication through the port.

Allowed_Connection_Binding: **inherit list of reference** (processor, virtual processor, bus, virtual bus, device, memory)

applies to (feature, connection, thread, thread group, process, system, virtual bus);

The Allowed_Connection_Binding property specifies a set of execution platform components to constrain the binding of connections and virtual buses. The named components must belong to a processor, virtual processor, bus, virtual bus, device, or memory category. When specified for a feature such as a port it indicates a binding constraint for all connections through that feature, e.g., any protocol assumptions a component makes about its communication through the port.

Actual_Connection_Binding: **inherit list of reference** (processor, virtual processor, bus, virtual bus, device, memory)

applies to (feature, connection, thread, thread group, process, system, virtual bus);

Connections and virtual buses are bound to the bus, virtual bus, processor, virtual processor, device, and memory specified by the Actual_Connection_Binding property. The entries in the list represent the flow sequence of the connection through the execution platform.

Allowed_Subprogram_Call: **list of reference** (subprogram)

applies to (subprogram access);

A subprogram call can be bound to any member of the set of subprograms specified by the Allowed_Subprogram_Call property. These may be remote subprograms, i.e., subprogram instances in other threads, or local subprogram instances. In the latter case the property identifies a specific code instance. If no value is specified, then subprogram call must be a local call.

Actual_Subprogram_Call: **reference** (subprogram)

applies to (subprogram access);

The Actual_Subprogram_Call property specifies the subprogram instance whose code is servicing the subprogram call. These may be remote subprograms, i.e., subprogram instances in other threads, or local subprogram instances. In the latter case the property identifies a specific local code instance, i.e., it can model sharing of subprogram. If no value is specified, the subprogram call is a local call.

Allowed_Subprogram_Call_Binding:
inherit list of reference (bus, processor, device)

applies to (subprogram, thread, thread group, process, system);

Remote subprogram calls can be bound to the physical connection of an execution platform that is specified by the Allowed_Subprogram_Call_Binding property. If no value is specified, then subprogram call may be a local call or the binding may be inferred from the bindings of the caller and callee.

Actual_Subprogram_Call_Binding: **inherit list of reference** (bus, processor, memory, device)

applies to (subprogram);

The Actual_Subprogram_Call_Binding property specifies the bus, processor, device, or memory to which a remote subprogram call is bound. If no value is specified, the subprogram call is a local call or the binding can be inferred from the binding of the caller and callee.

Provided_Virtual_Bus_Class : **inherit list of classifier** (virtual bus)

applies to (bus, virtual bus, processor, virtual processor, device, memory, system);

The Provided_Virtual_Bus_Class property specifies the set of virtual bus classifiers (protocols) supported by a bus, virtual bus, virtual processor, device, or processor. The property indicates that a component with a binding requirement for a virtual bus classifier can be bound to a component whose Provided_Virtual_Bus_Class property value includes the desired virtual bus classifier. Note that the component with this property is not required to have virtual bus subcomponents.

Required_Virtual_Bus_Class : **inherit list of classifier** (virtual bus)

applies to (virtual bus, connection, port);

The Required_Virtual_Bus_Class property specifies the set of virtual bus classifiers (protocols) that this connection or virtual bus needs to be bound to, i.e., that it requires to be bound to one instance of each of the specified classifiers. This property complements the Allowed_Connection_Binding_Class property, which specifies that the connection binding must be to components of one of the specified classifiers.

```
Provided_Connection_Quality_Of_Service : inherit list of
Supported_Connection_QoS
   applies to (bus, virtual bus, processor, virtual processor, system,
device, memory);
```

The Provided_Connection_Quality_Of_Service property specifies the quality of service provided by a protocol, i.e., a virtual bus, bus, virtual processor, or processor supporting protocols, for its transmission.

```
Required_Connection_Quality_Of_Service : inherit list of
Supported_Connection_QoS
   applies to (port, connection, virtual bus);
```

The Required_Connection_Quality_Of_Service property specifies that a connection or virtual bus expects a certain quality of service from the protocol that is used for its transmission.

```
Not_Collocated: record (
    Targets: list of reference (data, thread, process, system, connection);
    Location: classifier ( processor, memory, bus, system ); )
    applies to (process, system);
```

The Not_Collocated property specifies that hardware resources used by several software components must be distinct. The components referenced by Target must not be bound to the same hardware of the type specified in the Location field. If the Location is a system component, then they may not be collocated to any component contained in the system component.

```
Collocated: record (
    Targets: list of reference (data, thread, process, system, connection);
    Location: classifier ( processor, memory, bus, system ); )
    applies to (process, system);
```

The Collocated property specifies that several software components must be bound to the same hardware. The components referenced by Target must be bound to the same hardware of the type specified in the Location field. If the Location is a system component, then they must be collocated on any component contained in the system component.

(2) The next set of properties specify characteristics of the computing hardware as it relates to the deployment of software.

```
Allowed_Connection_Type: list of enumeration
                        (Sampled_Data_Connection, Immediate_Data_Connection,
                         Delayed_Data_Connection, Port_Connection,
                         Data_Access_Connection,
                         Subprogram_Access_Connection)
    applies to (bus, device);
```

The Allowed_Connection_Type property specifies the categories of connections a bus supports. That is, a connection may only be legally bound to a bus if the bus supports that category of connection.

If a list of allowed connection protocols is not specified for a bus, then any category of connection

can be bound to the bus.

Allowed_Dispatch_Protocol: **list of** Supported_Dispatch_Protocols

applies to (processor);

The Allowed_Dispatch_Protocol property specifies the thread dispatch protocols are supported by a processor. That is, a thread may only be legally bound to the processor if the specified thread dispatch protocol of the processor corresponds to the dispatch protocol required by the thread.

If a list of allowed scheduling protocols is not specified for a processor, then a thread with any dispatch protocol can be bound to and executed by the processor.

Allowed_Period: **list of** Time_Range

applies to (processor, system);

The Allowed_Period property specifies a set of allowed periods for periodic tasks bound to a processor.

The period of every thread bound to the processor must fall within one of the specified ranges.

If an allowed period is not specified for a processor, then there are no restrictions on the periods of threads bound to that processor.

Allowed_Physical_Access_Class: **list of classifier** (device, processor, memory, bus)

applies to (bus);

The Allowed_Physical_Access_Class property specifies the classifiers of processors, devices, memory, and buses that are allowed to be connected to the bus, i.e., whose connection is supported by the bus.

If the property is not specified for a bus, then the bus may be used to connect both devices and memory to the processor.

Allowed_Physical_Access: **list of reference** (device, processor, memory, bus)

applies to (bus);

The Allowed_Physical_Access property specifies the classifiers of processors, devices, memory, and buses that are allowed to be connected to the bus, i.e., whose connection is supported by the bus.

If the property is not specified for a bus, then the bus may be used to connect both devices and memory to the processor.

Memory_Protocol: **enumeration** (read_only, write_only, read_write) => read_write

applies to (memory);

The Memory_Protocol property specifies memory access and storage behaviors and restrictions. Writeable data produced from software source text may only be bound to memory components that have the write_only or read_write property value.

Scheduling_Protocol: **inherit list of** Supported_Scheduling_Protocols

applies to (virtual processor, processor);

The Scheduling_Protocol property specifies what scheduling protocol the thread scheduler of the processor uses. The core standard does not prescribe a particular scheduling protocol.

Scheduling protocols may result in schedulers that coordinate scheduling of threads across

multiple processors.

```
Preemptive_Scheduler : aadlboolean
        applies to (processor);
```

This property specifies if the processor can preempt a thread during its execution. By default, if this property is not specified, the processor owns a preemptive scheduler.

```
Thread_Limit: aadlinteger 0 .. Max_Thread_Limit
   applies to (processor);
```

The `Thread_Limit` property specifies the maximum number of threads supported by the processor.

```
Priority_Map: list of Priority_Mapping
   applies to (processor);
```

The `Priority_Map` property specifies a mapping of AADL priorities into priorities of the underlying real-time operating system. This map consists of a list of `aadlinteger` pairs.

```
Priority_Mapping: type record (
       Aadl_Priority: aadlinteger;
       RTOS_Priority: aadlinteger; )
   applies to (processor);
```

The `Priority_Mapping` property specifies a mapping of a single AADL priority value into a single priority value of the underlying real-time operating system. This property is used to define the elements of a consists of a `Priority_Map`.

```
Priority_Range: range of aadlinteger
   applies to (processor);
```

The `Priority_Range` property specifies the range of thread priority values that are acceptable to the processor.

The property type is `range of aadlinteger`.

end Deployment_Properties;

A.2 Predeclared Thread Properties

(1) The properties of the predeclared property set named `Thread_Properties` record information related to threads and devices, i.e., active application components. They address dispatching, concurrency, and mode transition.

Property set Thread_Properties **is**

```
Dispatch_Protocol: Supported_Dispatch_Protocols
   applies to (thread);
```

The Dispatch_Protocol property specifies the dispatch behavior for a thread.

A method used to construct a actual system from a specification is permitted to support only a subset of the standard scheduling protocols. A method used to construct a actual system is permitted to support additional non-standard scheduling protocols.

Dispatch_Trigger: **list of reference** (port)

applies to (device, thread);

The Dispatch_Trigger property specifies the list of ports that can trigger the dispatch of a thread or device.

POSIX_Scheduling_Policy : **enumeration** (SCHED_FIFO, SCHED_RR, SCHED_OTHERS)

applies to (thread, thread group);

The POSIX_Scheduling_Policy property is used for the modeling of the scheduling protocols defined by POSIX 1003.1b. Such a property specifies the policy assign to a given thread. The policy may be either SCHED_FIFO, SCHED_RR or SCHED_OTHER. In a POSIX 1003.1b architecture, the policy allows the scheduler to choose the thread to run when several threads have the same fixed priority. If a thread does not define the POSIX_Scheduling_Policy property, it has by default the SCHED_FIFO policy. The policy semantics are :

- SCHED_FIFO : this policy implements a FIFO scheduling protocol on the set of equal fixed priority : a thread stays on the processor until it has terminated or until a highest priority thread is released.
- SCHED_RR : this policy is similar to SCHED_FIFO except that the quantum is used. At the end of the quantum, the running thread is pre-empted from the processor and a equal priority thread has to be released.
- SCHED_OTHER : its semantic is defined by POSIX policy implementers. This policy usually implements a timing sharing scheduling protocol.

Priority: **inherit aadlinteger**

applies to (thread, thread group, process, system, device);

The Priority property specifies the priority of the thread that is taken into consideration by some scheduling protocols in scheduling the execution order of threads.

The property type is aadlinteger. Its value is expected to be within the range of priority values supported by a given processor.

Criticality: **aadlinteger**

applies to (thread, thread group);

This property specifies the criticality level of a thread. This property is used by maximum urgency first scheduling protocols. Such a property can also be used by any project specific scheduling protocols.

Time_Slot: **list of aadlinteger**

applies to (thread, thread group, process, virtual processor, system);

The Time_Slot property specifies statically allocated slots on a timeline. This property is used by scheduling protocols with a time slot allocation approach, such as the protocol for scheduling partitions on a static timeline.

Concurrency_Control_Protocol: Supported_Concurrency_Control_Protocols

applies to (data);

The Concurrency_Control_Protocol property specifies the concurrency control protocol used to ensure mutually exclusive access, i.e., a critical region, to a shared data component. If no value is specified the default value is None_Specified, i.e., no concurrency control protocol.

Urgency: **aadlinteger** 0 .. Max_Urgency

applies to (port, subprogram);

The Urgency property specifies the urgency with which an event at an in port is to be serviced relative to other events arriving at or queued at other in ports of the same thread. A numerically larger number represents higher urgency.

Dequeue_Protocol: **enumeration** (OneItem, MultipleItems, AllItems) => OneItem

applies to (event port, event data port);

The Dequeue_Protocol property specifies different dequeuing options.

- OneItem: (default) a single frozen item is dequeued at input time and made available to the source text unless the queue is empty. The Next_Value service call has no effect.
- AllItems: all items that are frozen at input time are dequeued and made available to the source text via the port variable, unless the queue is empty. Individual items become accessible as port variable value through the Next_Value service call. Any element in the frozen queue that are not retrieved through the Next_Value service call are discarded, i.e., are removed from the queue and are not available at the next input time.
- MultipleItems: multiple items can be dequeued one at a time from the frozen queue and made available to the source text via the port variable. One item is dequeued and its value made available via the port variable with each Next_Value service call. Any items not dequeued remain in the queue and are available at the next input time.
  If the Dequeued_Items property is set, then it imposes a maximum on the number of elements that are made accessible to a thread at input time when the Dequeue_Protocol property is set to AllItems or MultipleItems.

  The default property value is OneItem.

Dequeued_Items: **aadlinteger**

applies to (event port, event data port);

The Dequeued_Items property specifies the maximum number of items that are made available to the application via a port variable for event or event data ports when the input is frozen at input time. Its value cannot exceed that of the Queue_Size property for the same port. See also Dequeue_Protocol property.

Mode_Transition_Response: **enumeration** (emergency, planned)

applies to (mode transition);

The Mode_Transition_Response property specifies whether the mode transition occurs immediately due to an emergency, or whether it is planned in that the completion of thread execution can be coordinated before performing the mode transition. If not specified the mode transition is considered to be planned.

Resumption_Policy: **enumeration** (restart, resume)

applies to (thread, thread group, process, system, device, processor, memory, bus, system);

The Resumption_Policy property specifies whether as result of a mode transition activation a component that has modes itself starts in the initial mode or resumes in the current mode at the time of its deactivation.

Active_Thread_Handling_Protocol:

```
   inherit Supported_Active_Thread_Handling_Protocols => abort
   applies to (thread, thread group, process, system);
```

The Active_Thread_Handling_Protocol property specifies the protocol to use to handle execution at the time instant of an actual mode switch. The available choices are implementation defined. The default value is abort.

```
Active_Thread_Queue_Handling_Protocol:
   inherit enumeration (flush, hold) => flush
   applies to (thread, thread group, process, system);
```

The Active_Thread_Queue_Handling_Protocol property specifies the protocol to use to handle the content of any event port or event data port queue of a thread at the time instant of an actual mode switch. The available choices are flush and hold. Flush empties the queue. Hold keeps the content in the queue of the thread being deactivated until it is reactivated. The default value is flush.

```
Deactivation_Policy: enumeration (inactive, unload) => inactive
   applies to (process, virtual processor, processor);
```

The Deactivation_Policy property specifies whether a process is to be unloaded when it is deactivated. If the policy is unload, then the process is unloaded on deactivate and loaded on activate.
The default is that the process is loaded during startup and is not unloaded when deactivated.

```
Runtime_Protection : inherit aadlboolean
   applies to (process, system);
```

This property specifies whether a process requires runtime enforcement of address space protection. If no value is specified the default is assumed to be **true**.

```
Subprogram_Call_Type: enumeration (Synchronous, SemiSynchronous)
    => Synchronous
   applies to (subprogram);
```

The Subprogram_Call_Type property specifies whether the call is to be performed synchronous or semi-synchronous. In case of a semi-synchronous call the user of the result is may be suspended uintil the result is available. The default is Synchronous if no property value is specified.

```
Synchronized_Component: inherit aadlboolean => true
      applies to (thread, thread group, process, system);
```

The Synchronized_Component property specifies whether a periodic thread will be synchronized with transitions into and out of a mode. In other words, the thread affects the hyperperiod for mode switching of the property value is true.
The default value is true.

```
end Thread_Properties;
```

A.3 Predeclared Timing Properties

(2) The predeclared property set named Timing_Properties contains execution time related property definitions regarding threads, devices, and runtime system support for thread execution.

```
Property set Timing_Properties is
```

Time: **type aadlinteger** 0 ps .. Max_Time **units** Time_Units;

The Time property type specifies a property type for time that is expressed as numbers with predefined time units.

Time_Range: **type range of** Time;

The Time_Range property type specifies a property type for a closed range of time, i.e., a time span including the lower and upper bound.

The property type is Time. The standard units are ps (picoseconds), ns (nanoseconds), us (microseconds), ms (milliseconds), sec (seconds), min (minutes) and hr (hours).

(3) These properties record information related to the timing of thread and device execution timing.

Activate_Deadline: Time

applies to (Thread);

Activate_Deadline specifies the maximum amount of time allowed for the execution of a thread's activation sequence. The numeric value of time must be positive.

The property type is Time. The standard units are ps (picoseconds), ns (nanoseconds), us (microseconds), ms (milliseconds), sec (seconds), min (minutes) and hr (hours).

Activate_Execution_Time: Time_Range

applies to (thread);

Activate_Execution_Time specifies the minimum and maximum execution time, in the absence of runtime errors, that a thread will use to execute its activation sequence, i.e., when a thread becomes active as part of a mode switch. The specified execution time includes all time required to execute any service calls that are executed by a thread, but excludes any time spent by another thread executing remote procedure calls in response to a remote subprogram call made by this thread.

Compute_Deadline: Time

applies to (thread, device, subprogram, subprogram access, event port, event data port);

The Compute_Deadline specifies the maximum amount of time allowed for the execution of a thread's compute sequence. If the property is specified for a subprogram, event port, or event data port feature, then this compute execution time applies to the dispatched thread when the corresponding call, event, or event data arrives. When specified for a subprogram access feature, the Compute_Deadline applies to the thread executing the remote procedure call in response to the remote subprogram call. The Compute_Deadline specified for a feature must not exceed the Compute_Deadline of the associated thread. The numeric value of time must be positive.

The values specified for this property for a thread are bounds on the values specified for specific features.

The Deadline property places a limit on Compute_Deadline and Recover_Deadline: Compute_Deadline + Recover_Deadline ≤ Deadline.

The property type is Time. The standard units are ps (picoseconds), ns (nanoseconds), us (microseconds), ms (milliseconds), sec (seconds), min (minutes) and hr (hours).

Compute_Execution_Time: Time_Range

applies to (thread, device, subprogram, event port, event data port);

The Compute_Execution_Time property specifies the amount of time that a thread will execute after a thread has been dispatched, before that thread begins waiting for another dispatch. If the property is specified for a subprogram, event port, or event data port feature, then this compute execution time applies to the dispatched thread when the corresponding call, event, or event data initiates a dispatch. When specified for a subprogram (access) feature, it applies to the thread executing the remote procedure call in response to a remote subprogram call. The Compute_Execution_Time specified for a feature must not exceed the Compute_Execution_Time of the associated thread.

The range expression specifies a minimum and maximum execution time in the absence of runtime errors. The specified execution time includes all time required to execute any service calls that are executed by a thread, but excludes any time spent by another thread executing remote procedure calls in response to a remote subprogram call made by the thread.

The values specified for this property for a thread are bounds on the values specified for specific features.

Client_Subprogram_Execution_Time: Time_Range

applies to (subprogram);

The Client_Subprogram_Execution_Time property specifies the length of time it takes to execute the client portion of a remote subprogram call.

The property type is Time_Range. The standard units are ns (nanoseconds), us (microseconds), ms (milliseconds), sec (seconds), min (minutes) and hr (hours). The numeric value must be a positive number.

Deactivate_Deadline: Time

applies to (thread);

The Deactivate_Deadline property specifies the maximum amount of time allowed for the execution of a thread's deactivation sequence. The numeric value of time must be positive.

The property type is Time. The standard units are ps (picoseconds), ns (nanoseconds), us (microseconds), ms (milliseconds), sec (seconds), min (minutes) and hr (hours).

Deactivate_Execution_Time: Time_Range

applies to (thread);

The Deactivate_Execution_Time property specifies the amount of time that a thread will execute its deactivation sequence, i.e., when the thread is deactivated as part of a mode switch.

The range expression specifies a minimum and maximum execution time in the absence of runtime errors. The specified execution time includes all time required to execute any service calls that are executed by a thread, but excludes any time spent by another thread executing remote procedure calls in response to a remote subprogram call made by this thread.

Deadline: **inherit** Time => Period

applies to (thread, thread group, process, system, device);

The Deadline property specifies the maximum amount of time allowed between a thread dispatch and the time that thread begins waiting for another dispatch. Its numeric value must be positive.

The Deadline property places a limit on Compute_Deadline and Recover_Deadline: Compute_Deadline + Recover_Deadline ≤ Deadline

The Deadline property may not be specified for threads with background dispatch protocol.

`First_Dispatch_Time` **: inherit** `Time`

applies to `(thread, thread group);`

This property specifies the time of the first dispatch request.

`Dispatch_Jitter`: **inherit** `Time`

applies to `(thread, thread group);`

The `Dispatch_Jitter` property specifies a maximum bound on the lateness of a thread dispatching. In the case of a periodic thread for instance, the thread is supposed to be dispatched according to a fixed delay called the period. However, for many reasons, a periodic thread dispatching event can be delayed. The `Dispatch_Jitter` property can be used to specify such a delay. The `Dispatch_Jitter` property can be specified on any thread which can be dispatched several times (e.g.,. `Periodic`, `Sporadic`).

`Dispatch_Offset`: **inherit** `Time`

applies to `(thread);`

The `Dispatch_Offset` property specifies a dispatch time offset for a thread. The offset indicates the amount of clock time by which the dispatch of a thread is offset relative to its period. This property applies only to periodic threads.

`Execution_Time`: `Time`

applies to `(virtual processor);`

The `Execution_Time` property specifies the amount of execution time allocated to a virtual processor. This is the amount of execution time the virtual processor can make available to threads or virtual processors it schedules.

`Finalize_Deadline`: `Time`

applies to `(thread);`

The `Finalize_Deadline` property specifies the maximum amount of time allowed for the execution of a thread's finalization sequence. The numeric value of time must be positive.

The property type is `Time`. The standard units are ps (picoseconds), ns (nanoseconds), us (microseconds), ms (milliseconds), sec (seconds), min (minutes) and hr (hours).

`Finalize_Execution_Time`: `Time_Range`

applies to `(thread);`

The `Finalize_Execution_Time` property specifies the amount of time that a thread will execute its finalization sequence.

The range expression specifies a minimum and maximum execution time in the absence of runtime errors. The specified execution time includes all time required to execute any service calls that are executed by a thread, but excludes any time spent by another thread executing remote procedure calls in response to a remote subprogram call made by this thread.

`Initialize_Deadline`: `Time`

applies to `(thread);`

The `Initialize_Deadline` property specifies the maximum amount of time allowed between the time a thread executes its initialization sequence and the time that thread begins waiting for a

dispatch. The numeric value of time must be positive.

The property type is `Time`. The standard units are ps (picoseconds), ns (nanoseconds), us (microseconds), ms (milliseconds), sec (seconds), min (minutes) and hr (hours).

`Initialize_Execution_Time: Time_Range`

applies to `(thread);`

The `Initialize_Execution_Time` property specifies the amount of time that a thread will execute its initialization sequence.

The range expression specifies a minimum and maximum execution time in the absence of runtime errors. The specified execution time includes all time required to execute any service calls that are executed by a thread, but excludes any time spent by another thread executing remote procedure calls in response to a remote subprogram call made by this thread.

`Load_Deadline: Time`

applies to `(process, system);`

The `Load_Deadline` property specifies the maximum amount of elapsed time allowed between the time the process begins and completes loading. Its numeric value must be positive.

The property type is `Time`. The standard units are ns (nanoseconds), us (microseconds), ms (milliseconds), sec (seconds), min (minutes) and hr (hours).

`Load_Time: Time_Range`

applies to `(process, system);`

The `Load_Time` property specifies the amount of execution time that it will take to load the binary image associated with a process. The numeric value of time must be positive.

When applied to a system, the property specifies the amount of time it takes to load the binary image of data components declared within the system implementation and shared across processes (and their address spaces).

The range expression specifies a minimum and maximum load time in the absence of runtime errors.

`Period: inherit Time`

applies to `(thread, thread group, process, system, device, virtual processor);`

The `Period` property specifies the time interval between successive dispatches of a thread whose scheduling protocol is *periodic*, or the minimum interval between successive dispatches of a thread whose scheduling protocol is *sporadic*.

The property type is `Time`. The standard units are ns (nanoseconds), us (microseconds), ms (milliseconds), sec (seconds), min (minutes) and hr (hours). The numeric value must be a single positive number.

A `Period` property association is only allowed if the thread scheduling protocol is either `periodic` or `sporadic`.

`Recover_Deadline: Time`

applies to `(thread);`

`Recover_Deadline` specifies the maximum amount of time allowed between the time when a detected error occurs and the time a thread begins waiting for another dispatch. Its numeric value must be positive.

The `Recover_Deadline` property may not be specified for threads with `background` dispatch

protocol.

The Recover_Deadline must not be greater than the specified period for the thread, if any.

The Deadline property places a limit on Compute_Deadline and Recover_Deadline: Compute_Deadline + Recover_Deadline ≤ Deadline.

The property type is Time. The standard units are ps (picoseconds), ns (nanoseconds), us (microseconds), ms (milliseconds), sec (seconds), min (minutes) and hr (hours).

Recover_Execution_Time: Time_Range

applies to (thread);

The Recover_Execution_Time property specifies the amount of time that a thread will execute after an error has occurred, before it begins waiting for another dispatch.

The range expression specifies a minimum and maximum execution time in the absence of runtime errors. The specified execution time includes all time required to execute any service calls that are executed by a thread, but excludes any time spent by another thread executing remote procedure calls in response to a remote subprogram call made by this thread.

Startup_Deadline: Time

applies to (processor, virtual processor, process, system);

The Startup_Deadline property specifies the deadline for processor, virtual processor, process, and system initialization.

The property type is Time. The standard units are ps (picoseconds), ns (nanoseconds), us (microseconds), ms (milliseconds), sec (seconds), min (minutes) and hr (hours). The numeric value must be a single positive number.

Startup_Execution_Time: Time_Range

applies to (virtual processor, processor, process, system);

The Startup_Execution_Time property specifies the execution time for initialization of a virtual processor or process. Initialization time for threads is accounted for through its initialize entrypoint.

The property type is Time. The standard units are ps (picoseconds), ns (nanoseconds), us (microseconds), ms (milliseconds), sec (seconds), min (minutes) and hr (hours). The numeric value must be a single positive number.

(4) The following properties specify timing information related to the computing platform executing threads.

Clock_Jitter: Time

applies to (processor, system);

The Clock_Jitter property specifies a time unit value that gives the maximum time between the start of clock interrupt handling on any two processors in a multi-processor system.

The property type is Time. The standard units are ns (nanoseconds), us (microseconds), ms (milliseconds), sec (seconds), min (minutes) and hr (hours). The numeric value must be a positive number.

Clock_Period: Time

applies to (processor, system);

The Clock_Period property specifies a time unit value that gives the time interval between two

clock interrupts.

The property type is `Time`. The standard units are ns (nanoseconds), us (microseconds), ms (milliseconds), sec (seconds), min (minutes) and hr (hours). The numeric value must be a positive number.

`Clock_Period_Range: Time_Range`

applies to (processor, system);

The `Clock_Period_Range` property specifies a time range value that represents the minimum and maximum value assignable to the `Clock_Period` property.

`Process_Swap_Execution_Time: Time_Range`

applies to (processor);

The `Process_Swap_Execution_Time` property specifies the amount of execution time necessary to perform a context swap between two threads contained in different processes.

The range expression specifies a minimum and maximum swap time in the absence of runtime errors.

`Reference_Processor: inherit classifier ( processor )`

applies to (subprogram, subprogram group, thread, thread group, process, device, system);

The `Reference_Processor` property specifies the processor based on which the execution time is specified. When code is bound to a different processor type, the `Scaling_Factor` of that processor is used to determine the execution, unless a binding specific execution time value is associated.

`Scaling_Factor : inherit aadlreal`

applies to (processor, system);

This property specifies the speed of a processor with respect to a reference processor.

`Scheduler_Quantum : inherit Time`

applies to (processor);

This property specifies the quantum of a given processor. The quantum is a maximum bound on the time a thread can hold the processor without being preempted. A quantum is typically used in time sharing scheduling and in POSIX 1003.1b scheduling (with the SCHED_RR policy). The quantum can be used with any user-defined schedulers. If the quantum is not specified for a given processor, the quantum has a positive infinitesimal value.

`Thread_Swap_Execution_Time: Time_Range`

applies to (processor, system);

The `Thread_Swap_Execution_Time` property specifies the amount of execution time necessary for performing a context swap between two threads contained in the same process.

The range expression specifies a minimum and maximum swap time in the absence of runtime errors.

`Frame_Period: Time`

```
   applies to (processor, virtual processor);
```

The Frame_Period property specifies the time period of a major frame in a static scheduling protocol, such as a cyclic executive.

```
Slot_Time: Time
   applies to (processor, virtual processor);
```

The Slot_Time property specifies the time period of a slot in major frame in a static scheduling protocol, such as a cyclic executive, if the protocol uses fixed slot times.

```
end Timing_Properties;
```

A.4 Predeclared Communication Properties

(1) The predeclared property set named Communication_Properties defines communication related properties specify connection topology and queuing characteristics.

```
Property set Communication_Properties is
```

```
Fan_Out_Policy: enumeration (Broadcast, RoundRobin, Selective, OnDemand)
   applies to (port);
```

The Fan_Out_Policy property specifies how the output is distributed to multiple recipients of a port with multiple outgoing connections. Broadcast sends to all recipients, RoundRobin to one recipient at a time in order, Selective sends to one recipient based on data content, and OnDemand to the next recipient waiting on a port for dispatch.
Note that Broadcast, RoundRobin, and Selective pass on data and events without queuing it, while OnDemand requires a queue that is serviced by the recipients.

```
Connection_Pattern: list of Supported_Connection_Patterns
   applies to (connection);
```

The Connection_Pattern property specifies how an individual connection between arrays of ports looks like. If the property is not set the One_to_One pattern applies.

```
Connection_Set: list of Connection_Pair
   applies to (connection);
```

The Connection_Set property specifies a list of specific source element and destination element of a semantic connection by their array indices.

```
Connection_Pair: type record (
        src: list of aadlinteger;
        dst: list of aadlinteger;);
```

```
Overflow_Handling_Protocol: enumeration (DropOldest, DropNewest, Error)
       => DropOldest
   applies to (event port, event data port, subprogram access);
```

The Overflow_Handling_Protocol property specifies the runtime behavior of a thread when

an event arrives and the queue is full. `DropOldest` removes the oldest event from the queue and adds the new arrival. `DropNewest` ignores the newly arrived event. `Error` causes the thread's error recovery to be invoked. The default value is `DropOldest`.

`Queue_Processing_Protocol: Supported_Queue_Processing_Protocols => FIFO`

applies to (event port, event data port, subprogram access);

The `Queue_Processing_Protocol` property specifies the protocol for processing elements in the queue.

`Queue_Size:` **aadlinteger** `0 .. Max_Queue_Size => 1`

applies to (event port, event data port, subprogram access);

The `Queue_Size` property specifies the size of the queue for an event, event data port, of a subprogram access feature, and of a data component being shared via data access. In the case of a subprogram access it represents the queue for remote subprogram calls. In the case of a data component it represents the queue used in resource locking.

`Required_Connection :` **aadlboolean => true**

applies to (feature);

The `Required_Connection` property specifies whether the port or subprogram requires a connection. If the value of this property is false, then it is assumed that the component can function without this port or subprogram access feature being connected.

The default value is that a connection is required.

`Timing :` **enumeration** (sampled, immediate, delayed) **=>** sampled

applies to (port);

The `Timing` property specifies the timing of port connections. By default the interaction is `sampled`, i.e., the receiving component samples at dispatch or during execution.

The default value is that a connection is required.

`Transmission_Type:` **enumeration** (push, pull)

applies to (data port, port connection, bus, virtual bus);

The `Transmission_Type` property specifies whether the transmission across a data port connection is initiated by the sender (`push`) or by the receiver (`pull`). By default the transmission is initiated by the sender. A `pull` transmission type results in data being transmitted at the rate of the receiver. In the case of event data port or event ports, a `pull` transmission type results in events or event data queued with the sender to be transmitted upon receiver request.

When associated with a connection the property represents the transmission type the connection expects. When associated with a port the property represents the transmission type expected by the port. When associated with a bus or virtual bus the property represents the transmission type that is provided by the bus or protocol.

(2) The following communication properties specify input and output characteristics of port based communication.

`Input_Rate: Rate_Spec => ( Value_Range => 1.0 .. 1.0; Rate_Unit =>`

```
PerDispatch; Rate_Distribution => Fixed; )
```

applies to (port);

The Input_Rate property specifies the number of inputs per dispatch or per second of data, events, event data, or subprogram calls. If no Input_Rate is specified the default is one input per thread dispatch.

If no distribution function is specified it is assumed to be Fixed.

```
Input_Time: list of IO_Time_Spec => ( Time => Dispatch; Offset => 0.0 ns ..
0.0 ns;)
```

applies to (port);

The Input_Time property specifies the amount of execution time that can pass after dispatch before the input is frozen on a given port. The property value is a pair of Time a time range Offset. The default input time is Dispatch with zero Offset. A typical property value is a time offset in terms of Start.

```
IO_Time_Spec : type record (
        Offset : TimeRange;
        Time : IO_Reference_Time;
 );
```

The IO_Time_Spec property specifies the amount of execution time Offset relative to a Time at which input or output occurs. The value consists of a reference point and time range pair.

```
IO_Reference_Time : enumeration (Dispatch, Start, Completion, Deadline,
NoIO);
```

The IO_Reference_Time property specifies the time reference point to be used for specifying when input or output is available. The reference points are dispatch time (typically with zero time offset), start time (zero or more time), completion time (amount of execution time until completion), and deadline (typically with zero time offset). NoIO indicates that no I/O occurs.

```
Output_Rate: Rate_Spec => ( Value_Range => 1.0 .. 1.0; Rate_Unit =>
PerDispatch; Rate_Distribution => Fixed; )
```

applies to (port);

The Output_Rate property specifies the number of outputs per dispatch or per second of data, events, event data, or initiations of subprogram calls. The default is one output per thread dispatch and the default distribution is Fixed.

```
Output_Time: list of IO_Time_Spec => ( Time => Completion; Offset => 0.0 ns
.. 0.0 ns;)
```

applies to (port);

The Output_Time property specifies the amount of execution time until completion at which output becomes available. The property value is a pair of Time and Offset. The default Output_Time is Completion with zero Offset. For data ports with a delayed connection the default output time is Deadline.

```
Rate_Spec : type record (
        Value_Range : aadlreal ;
        Rate_Unit : units (PerSecond, PerDispatch);
        Rate_Distribution : Supported_Distributions;
);
```

The Rate_Spec property specifies the number of input or output occurrences per Rate_Unit,

i.e., per dispatch or per second. The default `Rate_Distribution` is `Fixed`.

```
Subprogram_Call_Rate: Rate_Spec => ( Value_Range => 1.0 .. 1.0; Rate_Unit =>
PerDispatch; Rate_Distribution => Fixed; )
```

applies to (subprogram access);

The `Subprogram_Call_Rate` property specifies the number of subprogram calls per dispatch or per second. The default is one call per thread dispatch and the default distribution is `Fixed`.

(3) The following are communication timing related properties.

```
Transmission_Time: record (
    Fixed: Time_Range;
    PerByte: Time_Range; )
```

applies to (bus);

The `Transmission_Time` property specifies the parameters for a linear model for the time interval between the start and end of a transmission of a sequence of N bytes onto a bus. The transmission time is the time between the transmission of the first bit of the message onto the bus and the transmission of the last bit of the message onto the bus. This time excludes arbitration, queuing, or any other times that are a function of how bus contention and scheduling are performed for a bus.

The associated expressions must evaluate to a record of two ranges of nonnegative numbers.

The time required to transmit a message of N Bytes over the bus is `Fixed + N * PerByte`, where `Fixed` is any number that falls in the first range specified and `PerByte` is any number that falls in the second range specified.

```
Actual_Latency: Time_Range
```

applies to (flow);

The `Actual_Latency` property specifies the actual latency as determined by the implementation of the end-to-end flow through semantic connections. Its numeric value must be positive.

The property type is `Time_Range`. The standard units are ns (nanoseconds), us (microseconds), ms (milliseconds), sec (seconds), min (minutes) and hr (hours).

```
Latency: Time_Range
```

applies to (flow, connection, bus);

The `Latency` property specifies the minimum and maximum amount of elapsed time allowed between the time the data or events enter the connection or flow and the time it exits. Its numeric value must be positive.

The property type is `Time`. The standard units are ns (nanoseconds), us (microseconds), ms (milliseconds), sec (seconds), min (minutes) and hr (hours).

end Communication_Properties;

A.5 Predeclared Memory Properties

(1) The predeclared property set named `Memory_Properties` defines properties related to memory as storage, memory and device access.

Property set Memory_Properties **is**

Size: **type aadlinteger** 0B .. Max_Memory_Size **units** Size_Units; Memory size as integers with predefined size units.
Size_Range: **type range of** Size; The `Size_Range` property specifies a closed range of memory size values, i.e., a memory size range including the lower and upper bound.

(2) These properties record information related to memory and access of data.

Access_Right : Access_Rights => read_write **applies to** (Data, Bus, Data Access, Bus Access); The `Access_Right` property specifies the form of access that is permitted for a component. If associated with a requires access clause it specifies the intended access to the component being accessed. If associated with a provides access clause it specifies the type of access that is permitted to the component for which access is provided. This access may be direct through read and write access or indirect through subprograms provided with the data type. The provided access `Access_Right` must not exceed the access right specified for the component itself. The required access `Access_Right` must not exceed the access right specified by the provides access or the component itself.
Access_Rights : **type enumeration** (read_only, write_only, read_write, by_method); The `Access_Rights` property type specifies the literals used to indicate the form of access that is permitted for a component.
Access_Time: **record** (First: IO_Time_Spec ; Last: IO_Time_Spec ;) => (First =>(Time => Start; Offset => 0.0 ns .. 0.0 ns;); Last => (Time => Completion; Offset => 0.0 ns .. 0.0 ns;);) **applies to** (data access); The `Access_Time` property specifies the range of execution time during which the data component is being accessed.
Allowed_Message_Size: Size_Range **applies to** (bus); The `Allowed_Message_Size` property specifies the allowed range of sizes for a block of data that can be transmitted by the bus hardware in a single transmission (in the absence of packetization). The expression defines the range of data message sizes, excluding any header or packetization

overheads added due to bus protocols, that can be sent in a single transmission over a bus. Messages whose sizes fall below this range will be padded. Messages whose sizes fall above this range must be broken into two or more separately transmitted packets.

```
Assign_Time: record (
    Fixed: Time_Range;
    PerByte: Time_Range; )
  applies to (processor);
```

The Assign_Time property specifies a time unit value used in a linear estimation of the execution time required to move a block of bytes on a particular processor. The time required is (Number_of_Bytes * PerByte) + Fixed

```
Base_Address: aadlinteger 0 .. Max_Base_Address
  applies to (memory, data, data access, port);
```

The Base_Address property specifies the address of the first word in the memory. The addresses used to access successive words of memory are Base_Address, Base_Address + Word_Space, ...Base_Address + (Word_Count-1) * Word_Space.

In the case of a memory component it indicates what its starting address is. In the case of a data component, data access, or port it indicates what its starting address is in the memory it is bound to.

```
Device_Register_Address: aadlinteger
  applies to (port, feature group);
```

The Device_Register_Address property specifies the address of a device register that is represented by a port associated with a device. This property is optional. Device ports may be represented by a source text variable as part of the device driver software.

```
Read_Time: record (
    Fixed: Time_Range;
    PerByte: Time_Range; )
  applies to (memory);
```

The Read_Time property specifies a time unit value used in a linear estimation of the execution time required to read a block of bytes from memory. The time required is (Number_of_Bytes * PerByte) + Fixed

```
Source_Code_Size: Size
  applies to (data, thread, thread group, process, system, subprogram,
processor, device);
```

The Source_Code_Size property specifies the size of the static code and read-only data that results when the associated source text is compiled, linked, bound and loaded in the final system.

The property type is Size. The standard units are bits, Bytes (bytes), KByte (kilobytes), MByte (megabytes) and GByte (gigabytes).

```
Source_Data_Size: Size
  applies to (data, subprogram, thread, thread group, process, system,
processor, device);
```

The Source_Data_Size property specifies the size of the readable and writeable data that results when the associated source text is compiled, linked, bound and loaded in the final system. In the case of data types, it specifies the maximum size required to hold a value of an instance of the data type.

The property type is Size. The standard units are bits, Bytes (bytes), KByte (kilobytes), MByte (megabytes) and GByte (gigabytes).

```
Source_Heap_Size: Size
   applies to (thread, subprogram);
```

The Source_Heap_Size property specifies the minimum and maximum heap size requirements of a thread or subprogram.

The property type is Size. The standard units are bits, Bytes (bytes), KByte (kilobytes), MByte (megabytes) and GByte (gigabytes).

```
Source_Stack_Size: Size
   applies to (thread, subprogram, processor, device);
```

The Source_Stack_Size property specifies the maximum size of the stack used by a processor executive, a device driver, a thread or a subprogram during execution.

The property type is Size. The standard units are bits, Bytes (bytes), KByte (kilobytes), MByte (megabytes) and GByte (gigabytes).

```
Byte_Count: aadlinteger 0 .. Max_Word_Count
   applies to (memory);
```

The Byte_Count property specifies the number of bytes in the memory.

```
Word_Size: Size => 8 bits
   applies to (memory);
```

The Word_Size property specifies the size of the smallest independently readable and writeable unit of storage in the memory.

The property type is Size. The standard units are bits, Bytes (bytes), KByte (kilobyte), MByte (megabyte) and GByte (gigabyte).

```
Word_Space: aadlinteger 1 .. Max_Word_Space => 1
   applies to (memory);
```

The Word_Space specifies the interval between successive addresses used for successive words of memory. A value greater than 1 means the addresses used to access words are not contiguous integers.
The default value is 1.

```
Write_Time: record (
    Fixed: Time_Range;
    PerByte: Time_Range; )
   applies to (memory);
```

The Write_Time property specifies a time unit value used in a linear estimation of the execution time required to write a block of bytes to memory. The time required is (Number_of_Bytes * PerByte) + Fixed