SAE Aerospace
An SAE International Group

AEROSPACE STANDARD

SAE AS5506

Issued 2004-11

ARCHITECTURE ANALYSIS & DESIGN LANGUAGE (AADL)

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. This subcommittee is chaired by Bruce Lewis (bruce.a.lewis@us.army.mil +1-256-876-3224), US Army Aviation and Missile Command (AMCOM) Software Engineering Directorate (SED), Acquisition Technology Division.

The starting point for the AADL standard development was MetaH, an architecture description language and non-commercialized supporting toolset, developed at Honeywell Technology Laboratories under the sponsorship of the US Defense Advanced Research Projects Agency (DARPA) and US Army Aviation and Missile Command (AMCOM).

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**

TABLE OF CONTENTS

Table of Figures

Foreword

This standard was prepared by the Society of Automotive Engineers (SAE) Avionics Systems Division (ASD) Embedded Computing Systems Committee (AS-2) Architecture Description Language (AS-2C) subcommittee.

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

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.

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

The AADL standard provides explicit support for extensions to the core language through the property extension mechanism for defining and integrating new properties and property sets. It also includes annex subclauses for the definition and integration of complementary sublanguages.

The AADL standard is aligned with Object Management Group (OMG) Unified Modeling Language (UML) through profiles for AADL as defined in the annexes.

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

The AADL standard provides guidelines for users to transition between AADL models and program source text written in Ada 95 (ISO/IEC 8652/1995 (E)) and C (ISO/IEC 9899:1999).

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

Introduction

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. The 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). The 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.

The 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.

Since the 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. The AADL specification language 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

The website at http://www.aadl.info is an information source regarding the SAE AADL standard. It makes available papers on the 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.

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

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

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.

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

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
```

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.

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

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

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 architecture model defined in AADL describes the properties and interfaces of components. Components fall into two major categories: those that represent the execution platform and those representing the application. The former is typified by processors, buses, and memory, the latter by application software modules. 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.

The language is applicable to systems that are:

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

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 extension sets. Extension sets may be proposed for inclusion in this standard. Such extensions will be defined as part of a new Annex appended to the standard.

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.

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 RTOS, can be modeled through software and execution platform components. The 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 physical systems implemented from those specifications.

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 the AADL.

1.1 Purpose/Extent

The purpose of the 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.

The 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.

The 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. The 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]. The 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 95 or C, or domain-specific modeling languages such as MatLab®/Simulink®[3].

The 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.

Rules of conformance are specified between specifications written in the 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.

The 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.

[2] VHDL is the “Very High Speed IC Hardware Description Language (Formerly Verilog 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.

1.2 Field of Application

The 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. The AADL may be extended to support other applications as the need arises.

1.3 Structure of Document

1.3.1 A Reader's Guide

This standard contains a number of sections, appendices, and annexes. The sections define the core AADL. The appendices provide additional information, both normative and informative about the core language. Annexes define extensions to the core AADL and provide guidelines and an interchange format to enable the transition of AADL models to other tools. Annexes may also provide information to clarify some of the underlying concepts incorporated into the AADL model.

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 the AADL are defined in Section 13. Appendix C , 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.

The core of the Architecture Analysis & Design Language 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 the AADL. 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.

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 the AADL that represent application system software components, i.e., data, subprogram, thread, thread group, and process.

Section 6, Execution Platform Components, defines those modeling elements of the AADL that model execution platform components, i.e., processor, memory, 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, i.e., ports, subprograms, and provided and required access to support modeling of shared access to data and buses.

Section 9, Connections and Flows, defines the constructs to express interaction between components in terms of connections between component features and in terms of flows through a sequence of components.

Section 10, 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 the AADL with new properties.

Section 11, Operational Modes, defines modes and mode transitions to support modeling of operational modes with mode-specific system configurations and property values.

Section 12, 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 13, 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.

The following Appendix sections complete the definition of the core AADL.

Appendix A , Predeclared Property Sets, contains the standard AADL set of predeclared properties.

Appendix B , Profiles and Extensions, contains profiles and extensions that have been approved by the standards body.

Appendix C , Glossary, contains a glossary of terms.

Appendix D , Syntax Summary, contains a summary of the syntax as defined in the sections of this document.

The Annex sections introduce additions and extensions to the core AADL. Annex F has been included in this release of the standard. Other Annexes will be part of the next release of the standard.

Annex A, Graphical AADL Notation, defines a graphical representation of the AADL.

Annex B, 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 introduces another graphical notation for AADL concepts.

Annex C, AADL Data Interchange Formats, defines an XML-based interchange format in form of an XMI meta model and an XML schema.

Annex D, Language Compliance and Application Program Interface, defines language-specific rules for source text to be compliant with an architecture specification. The initial version of this annex defines the language specific rules for Ada 95 and C and specifies the Ada 95 and C Application Program Interface to runtime service calls. AADL specifications and tools that process specifications are not required to support source text written in the Ada 95 or C language, but if they do so then they must comply with this annex.

Annex E, Error Model, defines the component and system compliance rules and semantics for AADL specifications that deal with safety and security aspects of a system. AADL specifications are not required to address these aspects of a system, but if they do then they must comply with this annex.

Annex F, Possible Tools, contains a description of tool support for the AADL.

The core language and the Annexes 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".

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.

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

Appendix C , Glossary

Appendix D , Syntax Summary

Annex F, Possible Tools.

1.3.2 Structure of Clauses and Subclauses

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.

Syntax

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

Naming rules define rules for names that represent defining identifiers and references to previously defined identifiers.

Legality Rules

Legality rules define restrictions on AADL specifications. Legality rules must be validated by AADL processing tools.

Standard Properties

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

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.

Processing Requirements and Permissions

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

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

1.4 Error, Exception, Anomaly and Compliance

The 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.

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. 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.

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.

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.

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.

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*.

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).

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.

An AADL specification is *compliant* with this standard if it satisfies all the syntactic and legality rules defined herein.

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.

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 physical system. A developer is permitted to classify such components as anomalous rather than noncompliant.

A tool that operates on AADL specifications is *compliant* with this 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 B.1). 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.

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

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.

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 ;
```

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

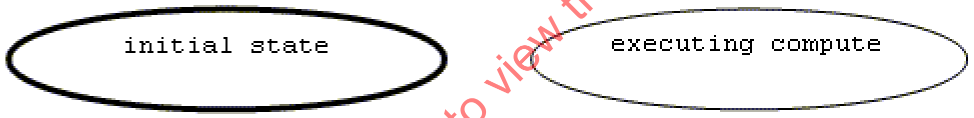
1.6 Method of Description for Discrete and Temporal Semantics

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.

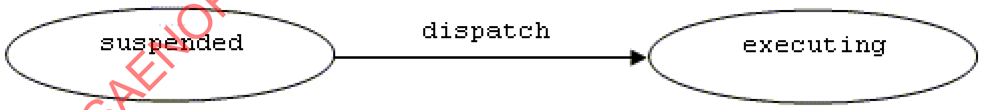
A semantic diagram defines the nominal scheduling and reconfiguration behavior for a modeled system as well as scheduling and reconfiguration behavior when failures are detected. A physical realization of a specification may violate this definition, for example due to runtime errors. A violation of the 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.

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.

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,

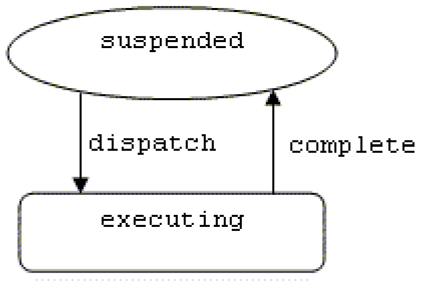


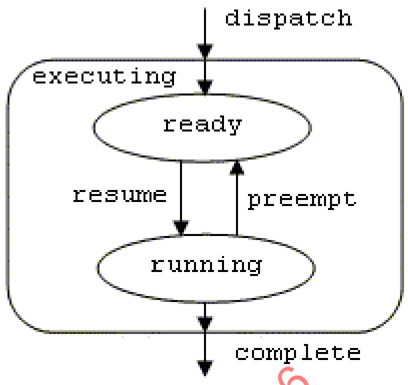
Directed edges labeled with one or more comma-separated, lower case phrases are used to denote possible transitions between the discrete states of a component. Transitions over an edge are logically instantaneous, i.e., the time interval in which a transition from a discrete state (called the source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



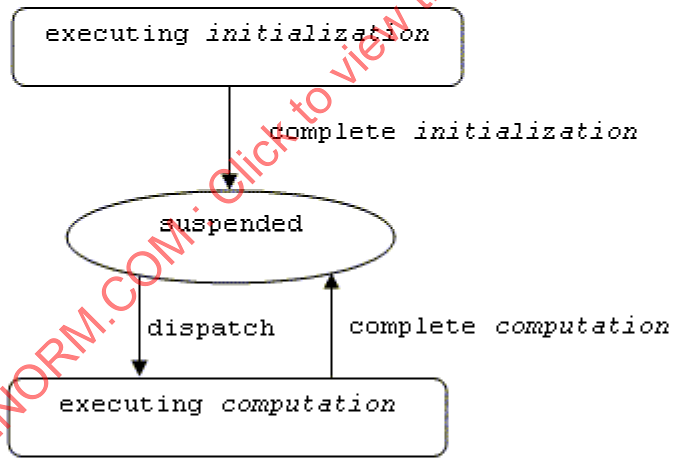
Permissions that allow a runtime implementation of a transition to occur over an interval of time are expressed as narrative rules. However, all implemented transitions must be atomic with respect to each other, all observable serializations must be admitted by the logical semantics, and all temporal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

Oblong boxes labeled with lower case phrases denote abstract discrete states that are defined as sets of other discrete states and edges. Wherever such an abstract discrete state appears in a hybrid semantics diagram, there will always be another hybrid semantics diagram showing an identically labeled oblong box that contains discrete states and edges to define that abstract discrete state. For example,

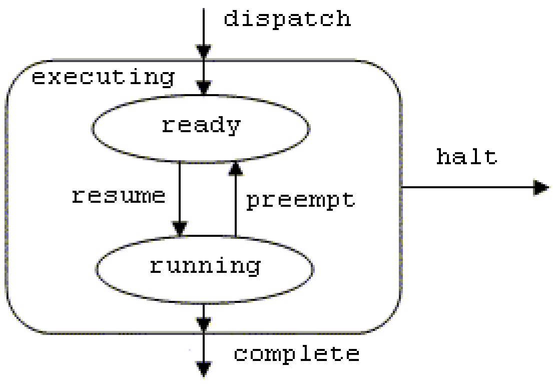




If there are multiple oblong boxes with the same label in a diagram, then multiple abstract discrete states are denoted. That is, the behavior is as if every occurrence of an abstract discrete state were replaced by a copy of its defining set of discrete states and transitions. In this standard, abstract states and edges that connect them will always be labeled so that the defining diagram for an abstract state, and the association between edges in the defining diagram and edges in the containing diagram, are unambiguous. 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`.



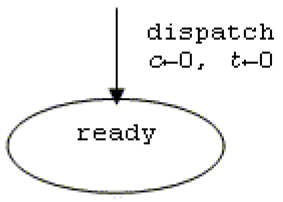
If there is an external edge that enters or exits the containing oblong box in the defining diagram for an abstract state, and there are no edges within that definition that connect any internal discrete state with that external edge, then there implicitly exist edges from every contained discrete state in the defining diagram to or from that external edge. That is, a transition over that external edge may occur for any discrete state in the defining diagram. For example, in the following diagram there is an implicitly defined `halt` edge out of both the `ready` and the `running` discrete states.



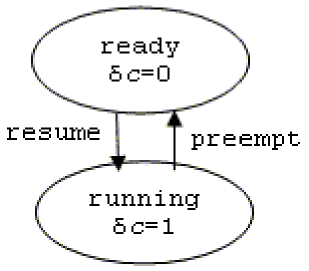
Real-valued variables whose values are time-varying may appear in expressions that annotate discrete states and edges of hybrid semantic diagrams. Specific forms of annotation are defined in subsequent paragraphs. The set of real-valued variables associated with a semantic diagram are those that appear 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.

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`.

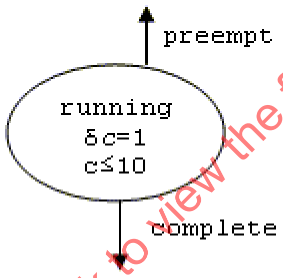
Edges may be annotated with assignments of values to variables associated with the semantic diagram. When a transition occurs over an edge, the values of the variables are set to the assigned values. For example, in the following diagram, the values of the variables *c* and *t* are set to 0 when the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



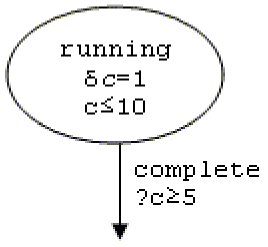
Discrete states may be annotated with expressions that define the possible rates of change for real-valued variables during the duration of time a component is in that discrete state. The rate of a variable is denoted using the symbol δ, for example δ*x*=[0,1] (the rate of the variable *x* may be any 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.



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.

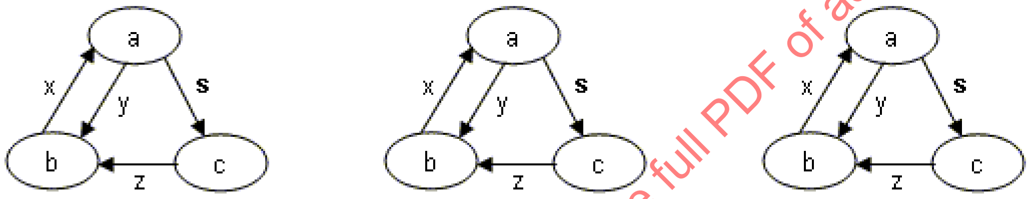


An edge may be annotated with Boolean-valued expressions called guards of that edge. A transition may occur from a source discrete state to a destination discrete state only when the values of the variables satisfy all guards for an edge between those discrete states. A guard on an edge is evaluated 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).



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.

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`.



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

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

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

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].

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.

ISO/IEC 8652:1995, Information Technology – Programming Languages – Ada.

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

Unified Modeling Language Specification [UML 2004, version 1.4.2], July 2004, version 1.4.2.

2.2 Informative References

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

[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.

[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.

[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

Terms are introduced throughout this standard, indicated by *italic* type. Informational definitions of terms are given in Appendix C , 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

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.

An *AADL specification* consists of *AADL global declarations* and *AADL declarations*. The AADL global declarations are comprised of *package specifications* and *property set declarations*. AADL declarations include *component type*s, *component implementations*, *port group types*, and *annex libraries*. AADL component type and implementation declarations model kinds of physical system components, such as a kind of hardware processor or a software program. This standard defines the following categories of components: *data*, *subprogram*, *thread*, *thread group*, *process*, *memory*, *bus*, *processor*, *device*, and *system*. They form the core of the AADL modeling vocabulary.

A component type specifies a functional interface in terms of *features*, *flow specifications*, and *properties*. It represents a specification of the component against which other components can operate. Implementations of the component are required to satisfy this specification.

A component implementation specifies an internal structure in terms of *subcomponents*, *connections* between the features of those subcomponents, *flows* across a sequence of subcomponents, *modes* to represent operational states, and properties. Unlike many other languages, the AADL allows multiple implementations to be declared with the same functional interface.

Packages provide a library-like structure for organizing component type and component implementation declarations into separate namespaces and combining them into a system specification.

Components may be hierarchically decomposed into collections or assemblies of interacting subcomponents. A subcomponent declares a component that is contained in another component, naming a component type and component implementation to specify an interface and implementation for the subcomponent. Thus, component types and implementations act as *component classifiers*. The hierarchy of a system instance is based upon the set of subcomponents of the top-level system implementation. It is completed by iteratively traversing the tree of the component classifiers specified starting at the top-level system implementation subcomponents.

A feature describes a functional interface of a component through which control and data may be exchanged with other components. Features can be *ports* to support directional flow of control and data, *subprograms* to represent synchronous procedure calls, and *requires* and *provides access* to subcomponents to represent shared access to data and bus components. Required subcomponent access specifies the need for a component to access components declared outside the component. Provided subcomponent access specifies that a subcomponent contained in a component is made externally accessible. Ports in an AADL specification may map to a variable in a piece of source code, i.e., a storage location in a physical memory.

Subcomponents allow systems to be specified as a static and tree-like containment hierarchy. The AADL also allows components to reference subcomponents that are not contained exclusively in the component. This allows a component to be accessed or used in more than one component. In the AADL, data and bus components can have shared access. For example, static data items contained in a source text software package and represented in AADL as data components may be used by threads in different processes (whose protected address spaces may otherwise be distinct).

Syntactically the terms component type declaration, component implementation declaration, and subcomponent declaration refer to specific grammar rules for each component category. Semantically, a component may have subcomponents while it itself is a subcomponent of some other component. The terms component and subcomponent must be interpreted semantically as a relationship between two components that are identified by context.

Components, features, modes, connections, flows, and subprogram calls can have properties. A property has a name, a type and a value. 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. Properties can have mode-specific and binding-specific values.

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. Property values can be associated with component types, component implementations, subcomponents, features, connections, flows, modes, and subprogram calls. For example, a property is used to identify the source code files associated with a software component. Another example of the use of properties is specifying hardware memory, i.e., the number of addressable storage units and their size.

AADL component type and component implementation declarations can be organized into packages. Each package provides a separate namespace for component type and implementation declarations. A component classifier in a package is referenced externally be qualifiing its name with the package name. Packages can be nested and referenced externally utilizing qualified names.

Features and flow specifications of component types may be partially specified. Similarly, subcomponents, connections, flows, and modes of component implementations may have incomplete specifications. These specifications may be later refined in component type and component implementation extensions with the completion of classifier references and property associations. Component type extensions can also introduce additional features, flow specifications, and properties. Such extensions can add new subcomponents, connections, flows, modes, and properties to component implementations.

A system modeled in AADL consists of application software mapped to an execution platform. Data, subprograms, threads, thread groups, and processes collectively represent application software. They are called *software* components. Processor, memory, bus, and device collectively represent the execution platform. They are called *execution platform* components. Execution platform components support the execution of threads, the storage of data and code, and the communication between threads. Systems are called *compositional* components. They permit software and execution platform components to be organized into hierarchical structures with well-defined interfaces. Operating systems may be represented through properties of the execution platform or, requiring significantly more detail, modeled as software components.

Software components model *source text, virtual address spaces,* and *units of concurrent execution.* Source text can be written in a programming language such as Ada 95, 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.

AADL data components represent static data in source text. This data can be shared by threads and processes; they do so by the 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 are documented in an appropriate implementation Annex in this standard.

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 subprogram component models source text that is executed sequentially. Subprograms are callable from within threads and subprograms. Subprograms may require access to data components.

AADL thread components model units of concurrent execution, i.e., concurrent runtime threads of control or threads of execution 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. 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, and aperiodic threads as well as background threads. Additional dispatch protocols may be defined. Threads can contain subprogram and data components, and provide or require access to data components.

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.

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 enforced at runtime 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 units of concurrent execution, they must contain at least one thread. Processes can contain thread groups, threads, and data components, and can access or share data components.

Execution platform components represent hardware and software that is capable of scheduling threads, of enforcing specified address space protection at runtime, of storing source text code and data, of interfacing with an external environment, and of performing communication for application system connections.

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.

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.

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.

AADL device components model physical devices that interface with an external environment, e.g. sensors and actuators providing an interface between a physical plant and a control system or a GPS system. They may exhibit complex behaviors. Devices are logically connected to application software components and physically connected to processors. They cannot store nor execute application software source text themselves, but may include driver software executed on a connected processor. A device requires access to buses.

AADL systems model hierarchical compositions of software and execution platform components. A system may contain data, thread, thread group, process, memory, processor, bus, device, and system subcomponents. A system may require and provide access to data and bus components.

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.

This standard defines several categories of features: data port, event port, event data port, port group, data subprogram, server subprogram, and subprogram parameter, and provided and required subcomponent access. 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. Ports groups support grouping of ports, such that they can be connected to other components through a single connection. Data subprograms represent entrypoints to code sequences in source text that are associated with a data type. Server subprograms represent connection points for synchronous call/returns between threads; in some instances the call/return may be remote. 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.

AADL connections specify patterns of control and data flow between individual components at runtime. A semantic connection can be made between two threads, between a thread and a device or processor, or between a thread, device, or processor and a mode transition. A mode transition 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 Annex A.

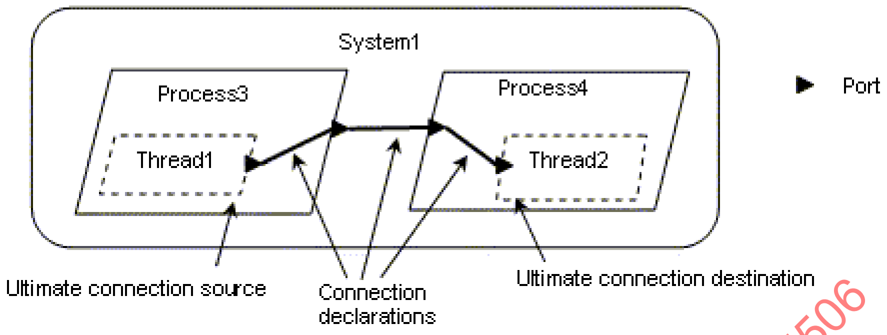


Figure 1 Example Semantic Connections

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. 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.

Flows describe actual flow sequences through components and sets of components across one or more connections. They are declared in component implementations. Flow sequences take two forms: *flow implementation* and *end-to-end flow*. A flow implementation describes how a flow specification of a component is realized in its component implementation. 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.

A physical system is modeled by instantiating a system implementation that consists of subcomponents representing the application software and execution platform components used to execute the application, including devices that interface with the external environment. A system instance represents the complete component hierarchy as specified by the system classifier's subcomponents and the subcomponents of their component classifiers down to the lowest level defined in the architecture specification.

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 physical 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.

The AADL core language is extensible through property sets, *annex subclauses* and *annex libraries*. 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 can be referenced in annex subclauses.

4 Components, Packages, and Annexes

The AADL defines the following *categories* of components: *data, subprogram*, *thread, thread group, process, memory, bus, processor, device,* and *system*. This section describes those aspects of components that are common to all AADL component categories. This section also describes packages as an organizing mechanism. This section closes with the definition of annex subclauses and annex libraries.

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.

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 (server subprograms). 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.

A component has zero or more *component implementations.* A component implementation specifies an internal structure for a component as an assembly of subcomponents. Subcomponents are instantiations of *component classifiers*, i.e., component types and implementations.

Components are named and have properties. These properties have associated expressions and values that represent attributes and behaviors of a component.

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 a common basis for the evolution of a family of related component types and implementations.

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

An AADL specification is a set of declarations: component classifier, port group classifier, annex library, package, and property-set. Package and property set declarations are global declarations. The content of global declarations can be referenced by any declaration. Component classifiers, port group types, and annex libraries that are declared directly in an AADL specification are anonymous declarations. They can only be referenced by another anonymous declaration.

Packages provide a way for organizing collections of component classifier, port group type, annex library declarations along with relevant property associations.

Property sets provide extensions to the core AADL that support additional modeling and analysis capabilities.

System instances are identified to processing tools and methodologies by referencing a system implementation component as the root of the system instance (see Section 12.1).

Syntax

```
AADL_specification ::=
    { AADL_global_declaration | AADL_declaration }+

AADL_global_declaration ::= package_spec | property_set


AADL_declaration ::=
    component_classifier
    | port_group_classifier
    | annex_library


component_classifier ::=
    component_type | component_type_extension |
    component_implementation | component_implementation_extension


port_group_classifier ::=
        port_group_type | port_group_type_extension
```

Naming Rules

The AADL has one *global* namespace. The package and property set identifiers comprising this space must be unique. These identifiers qualify the names of individual elements when they are referenced externally. They can be referenced from other declarations and anonymous declarations (see below). Package declarations represent labeled namespaces for component type, component implementation, port group type, and annex library declarations. Property sets represent labeled namespaces for property type and property name declarations.

An AADL specification has one *anonymous* namespace. In this are found the identifiers of component classifiers, port group classifiers, and annex libraries that are declared directly in an AADL specification. These identifiers must be unique in the anonymous namespace. Declarations of component classifiers and port group types can be referenced from other component classifier declarations in the anonymous namespace. Any annex library items declared in the anonymous namespace are only accessible from annex subclauses in component classifiers in the anonymous namespace.

AADL declarations in an AADL specification can refer to packages and property sets that may be separately stored. Those packages and property sets are considered to be part of the global namespace.

Defining identifiers in AADL must not be one of the reserved words of the language (see Section 13.7).

The AADL identifiers and reserved words can be in upper or lower case (or a mixture of the two) (see Section 13).

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

Semantics

An AADL specification provides a global namespace for packages and property sets and an anonymous namespace for component types, component implementations, annex libraries, and port group types. Items in the global namespace and their content can be named by items in the global and in the anonymous namespace. Items in the anonymous namespace can only be named by items in the anonymous namespace.

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 runing on different execution platforms. Packages allow such declarations to be organized into separate namespaces.

Port group types provide the definition of an interface to a component that represents a collection of ports or port groups defined within the component implementation (see Section 8.2). This group of ports may be accessed externally as a single unit.

A property set is used to introduce new property types and properties (see section 10.1). They extend the predefined set of properties of the core AADL.

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

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

A package provides a way to organize component types, component implementations, port group types, and annex libraries into related sets of declarations by introducing separate namespaces. Package names built using identifiers 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.

Syntax

```
package_spec ::=
    package defining_package_name
     ( public package_declaration [ private package_declaration ]
       | private package_declaration )
    end defining_package_name ;


package_declaration ::= { aadl_declaration }+
                         [ properties ( { property_association }+ |
                                          none_statement ) ]


package_name ::=
    { package_identifier :: }* package_identifier


none_statement ::= none ;
```

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

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. This means that the first identifier in a package name must be unique in the global namespace. Succeeding identifiers in the package name must be unique within the scope of the previous identifier. The **public** and **private** section of a package may be declared in separate package declarations; these two declarations introduce a single defining package name.

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, port 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.

The package namespace is divided into a public part and a private part. Items declared in the public part of the package namespace can be referenced from outside the package as well as within the package. Items declared in the private part of the package can only be referenced from within the public and private part of the package.

The reference to an item declared in another package must be an item *name* qualified with a package name separated by a double colon ("::"). Only the public package namespace is used to resolve these references. If the qualifying package identifier is missing, the referenced component classifier, port group type, or item in an annex library must exist in the same package, or in case of references from declarations in the AADL specification itself the referenced item must exist in the anonymous namespace. Component types, component implementations, port group types, and items in annex libraries declared directly in the AADL specification, i.e., the anonymous namespace, can only be referenced by other declarations in the AADL specification itself.

Legality Rules

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

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.

Semantics

A package provides a way to organize component type declarations, component implementation declarations, port group types, and annex libraries into related sets of declarations along with relevant property associations. It provides a namespace for component types, port 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 implemenation 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.

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 AADL specifications. Declarations in the **private** part are visible only within the package, i.e., names declared in the **private** part can only be referenced by declarations within the package.

4.3 Component Types

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, port groups, data components contained in the component that are made externally accessible, required access to externally provided components, and subprograms that are execution entrypoints to the component along with parameter declarations for the specification of the data values that flow into and out of subprograms. The ports and subprograms 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 or bus, is resolved when subcomponents of this component type are declared.

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.

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 and by associating new values with properties.

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 Annex A) 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*.

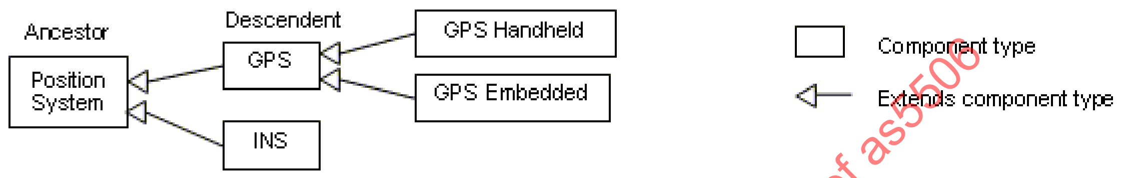


Figure 2 Component Type Extension Hierarchy

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 4.6)

Syntax

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


component_type_extension ::=
  component_category defining_component_type_identifier
    extends unique_component_type_identifier
  [ features ( { feature | feature_refinement }+ | none_statement ) ]
  [ flows ( { flow_spec | flow_spec_refinement }+ | none_statement ) ]
  [ properties ( { component_type_property_association }+ | none_statement ) ]
  { annex_subclause }*
  end defining_component_type_identifier ;


component_category ::=
             software_category
```

```
            | execution_platform_category
            | composite_category

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

execution_platform_category ::= memory | processor | bus | device

composite_category ::= system

unique_component_type_identifier ::=
    [ 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 **features**, **flows**, 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 ;**).

The annex_subclause of the component type is optional.

Naming Rules

The defining identifier for a component type must be unique within the anonymous namespace or within the package namespace of the package within which it is declared.

Each component type has an *interface namespace* for defining identifiers of features and flow specifications. That is, defining feature and defining flow specification identifiers must be unique in the interface namespace.

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, or in the anonymous namespace if the extension is declared in the AADL specification directly.

When a component type **extends** another component type, a component type interface namespace includes all the identifiers in the interface namespaces of its ancestors.

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

The defining identifier of a feature must be unique in the interface namespace of the component type.

The refinement identifier of a feature refinement refers to the closest refinement or the defining declaration of the feature going up the component type ancestor hierarchy.

Legality Rules

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

The **features**, **flows**, 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.

A component type declaration that does not extend another component type must not contain feature refinement declarations.

The category of the component type being extended must match the category of the extending component type.

Semantics

A component type represents the interface specification of a component, i.e., the component category, the features of a component, and property values. A component implementation denotes a component, existing or potential, that is compliant with the category, features and required subcomponents and properties declared for components of that type. 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.

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

Features of a component are interaction points with other components, i.e., ports and port groups; server subprograms, subprograms and parameters; required subcomponent access; and provided subcomponent access. Ports, port groups and subprograms specify both incoming and outgoing interaction points. Required subcomponent access declarations represent references to components that are not contained in the current component but must be accessed by the component. If accessed by multiple components they become shared components. Ports, port groups, subprograms, provided and required subcomponent access are described in Section 8.

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.

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. The use of incomplete declarations is particularly useful during early design stages where details may not be known or decided.

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 contain refinenment declarations 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, and property associations. This supports evolutionary development and modeling of system families by declaring partially complete component types that get refined in extensions.

Properties are predefined for each of the component categories and will be described in the appropriate sections. See Section 10.3 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;
```

4.4 Component Implementations

A *component implementation* contains subcomponents and their connections, properties, and component modes. Every component implementation is associated with a component type. A component type may have zero or more component implementations declared.

A component implementation consists of a collection of zero or more subcomponent and subcomponent refinements, connection and connection refinements, subprogram call sequences, component type feature refinements, flow sequences, and mode declarations; and zero or more property associations. 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.

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.

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.

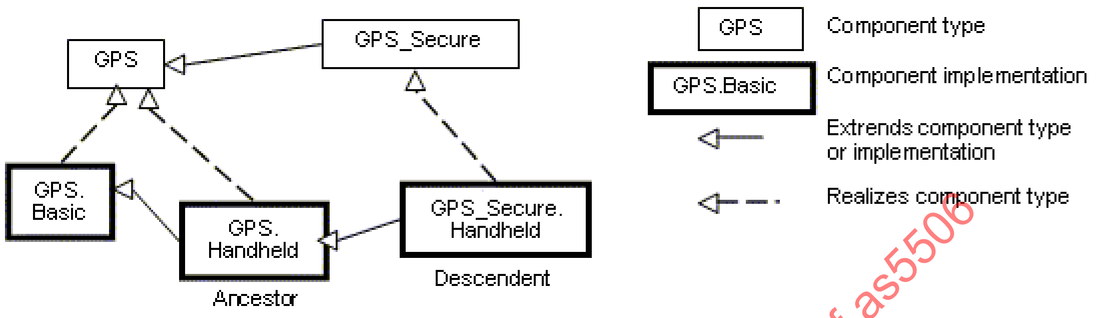


Figure 3 Extension Hierarchy of Component Types and Implementations

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 4.6).

Syntax

```
component_implementation ::=
   component_category implementation
   defining_component_implementation_name
   [ refines type ( { feature_refinement }+  | 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 ( { mode }+ { mode_transition }* | none_statement ) ]
   [ 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_name
[ refines type
     ( { feature_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 ( { mode | mode_refinement | mode_transition }+ | none_statement ) ]
[ properties ( { property_association }+  | none_statement ) ]
{ annex_subclause }*
end   defining_component_implementation_name ;
```

```
unique_component_implementation_name ::=
    [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 **refines type**, **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.

Naming Rules

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 or anonymous namespace as the component implementation.

The defining name of the component implementation must be unique within the anonymous namespace or within the package namespace of the package within which it is declared.

Every component implementation defines a *local namespace* for all defining identifiers of subcomponents, subprogram calls, connections, flows, and modes declared within the component implementation. The defining identifier of a 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.

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

Refinement identifiers of features must exist in the interface 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.

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.

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 interface namespaces).

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 subcomponents, connections, and modes, as well as to required subcomponents and features declared in the associated component type.

Legality Rules

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**.

The **refines type**, **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.

The category of the component implementation must match the category of the component type for which the component implementation is declared.

If the component implementation extends another component implementation, the category of both must match.

The **refines type** subclause must only contain refinement declarations of features in the component type and those refinements are limited to property associations. Specifically, the **refines type** subclause of a component implementation may not alter the component classifiers of inherited features.

Semantics

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.

Each component implementation is associated with a component type and provides a realization of its features (interface). A component type can have multiple implementations.

The physical 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 physical system structure.

A component implementation that is an extension of another inherits all subcomponents, connections, subprogram call sequences, flow sequences (flow implementations and end-to-end flows), modes, property associations, and annex subclauses from its ancestors as well as features, property associations, and annex subclauses from its associated component type (and that component type's ancestors).

A component implementation extension can also refine subcomponents previously declared in ancestor component implementations by completing 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 extension can also add 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.

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.

The **refines type** subclause of a component implementation can refine the property associations of features of its associated component type and of that component's ancestor component types. The example given in the section below illustrates the use of **refines type** to provide mappings of ports to source text variable names in different component implementations.

Properties are predefined for each of the component categories and will be described in the appropriate sections. See Section 10.3 regarding rules for determining property values.

NOTES:

Component implementation declarations can only refer to component types residing in the same package namespace. In order to add an implementation to a component type declared in another package, the component type can be created in the current namespace (package) by referencing the original package in a type extension in the current namespace. In the following example, LM::GPS is a reference to the original type defined in the package LM.

system GPS **extends** LM::GPS **end** GPS;

Processing Requirements and Permissions

A component implementation denotes a set of physical system components, existing or potential, that are compliant with the component implementation declaration as well as the associated component type. That is, the physical components denoted by a component implementation declaration are always compliant with the functional interface specified by the associated component type declaration. Physical 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.

In general, two physical components that comply with the same component type and component implementation are not necessarily substitutable for each other in a physical system. This is because an AADL specification may be legal but not specify all of the characteristics that are required to insure 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 physical system.

Examples

```
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
refines type
   CruiseActive: refined to out data port Bool_Type
                                { Source_Name => "CruiseControlActive"; };
properties
  Dispatch_Protocol=>Periodic;
  Period=> 10 ms;
end DriverModeLogic.Simulink;

thread implementation DriverModeLogic.C
refines type
  CruiseActive: refined to out data port Bool_Type
      { Source_Name => "CCActive"; };
properties
```

```
  Dispatch_Protocol=>Periodic;
  Period=> 10 ms;
end DriverModeLogic.C;
```

4.5 Subcomponents

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 physical system as a system instance. Figure 4 provides an illustration of a containment hierarchy using the graphical AADL notation (see Annex A). 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.

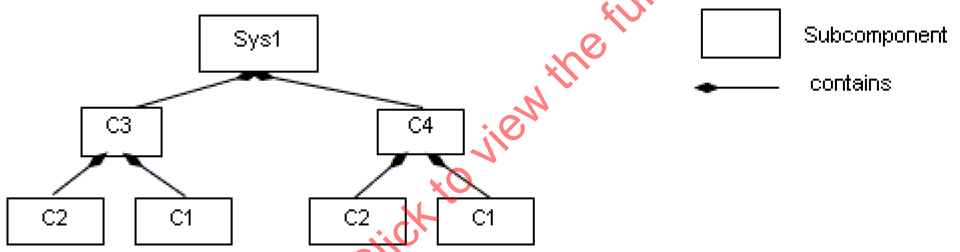


Figure 4 Component Containment Hierarchy

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.

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

Subcomponents can be refined as part of component implementation extensions. Refinement allows classifier references to be completed, and subcomponent property values to be associated. The resulting refined subcomponents can be refined themselves.

Syntax

```
subcomponent ::=
    defining_subcomponent_identifier :
       component_category [ component_classifier_reference ]
       [ { { subcomponent_property_association
            | contained_property_association }+ } ]
        [ in_modes ] ;
```

```
subcomponent_refinement ::=
    defining_subcomponent_identifier : refined to
       component_category [ component_classifier_reference ]
       [ { { subcomponent_property_association
              | contained_property_association }+ } ]
        [ in_modes ] ;

component_classifier_reference ::=
            unique_component_type_name [ . component_implementation_name ]
```

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

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.

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

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

NOTES:

The `Sample_Manager` in the example section below illustrates each kind of resolution.

Legality Rules

The category of the subcomponent declaration must be identical to the category its corresponding component classifier reference.

The component type named in the component classifier reference of a subcomponent refinement must be the component type of the subcomponent being refined if the subcomponent being refined has a component type declared. The component implementation named in the component classifier reference of a subcomponent refinement must be the component implementation of the subcomponent being refined if the subcomponent being refined has a component type declared.

If the subcomponent declaration contains an `in_modes` statement and any of its property associations also contains an `in_modes` statement, then the modes named in the property association must be a subset of those named in the subcomponent declaration. For more detail on the semantics of `in_modes` statements see Section 11.1.

Semantics

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 physical system.

A component implementation can contain *incomplete* subcomponent declarations, i.e., subcomponent declarations with no component classifier references or the component classifier reference only consists of a component type name for a component type with more than one component implementation. 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.

The optional `in_modes` subclause specifies the modes in which the subcomponent is active.

A subcomponent can have property associations for its own properties, or a contained property association for the properties of its subcomponents and their subcomponents, as well as those subcomponents' features, modes, subprogram call sequences, connections, and flows (see Section 10.3). Subcomponent refinements may declare property associations – that override the property values declared in the subcomponent being refined.

NOTES:

The example below illustrates the use of component type only as data component classifier. This is sufficient for implementation methods to perform analysis and to generate a physical system implementation from the AADL specification. In case of process components, the process component classifier reference must refer to a process implementation if the implementation method must process the complete system instance, e.g., performs scheduling analysis. In other words, some implementation methods and component categories require component classifier references to component implementations, while for others the component type reference is sufficient.

Examples

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. In that context it illustrates the ways of resolving required access.

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

   data Sample_Set
   properties
      Source_Data_Size => 1 MB;
   end Sample_Set;
   data Dynamic_Sample_Set extends Sample_Set
   end Dynamic_Sample_Set;
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;
end Collect_Samples;

thread implementation Collect_Samples.Batch_Update
refines type
   Input_Sample: refined to
          in event data port Sampling::Sample {Source_Name => "InSample"; };
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;

connections
   data access Samples -> Init_Samples.SampleSet;
   data access External_Samples -> Init_Samples.OrigSet;
   data access Samples -> Collect_Samples.SampleSet;
   event data port Input_Sample -> Collect_Samples.Input_Sample;
   data access Samples -> Distribute.SampleSet;
   event data port Distribute.UpdatedSamples -> Result_Sample;
end Sample_Manager.Slow_Update;
```

4.6 Annex Subclauses and Annex Libraries

Annex subclauses contain declarations expressed in a sublanguage that can be added to component types and component implementations through annexes. Examples of annex subclauses are assertions.

Annex libraries are reusable declarations expressed in a sublanguage that are declared in packages. Those reusable declarations can be referenced by annex subclauses.

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

The AADL standard consists of a core language and a set of approved annexes. An AADL specification is compliant with the standard if it restricts itself to the core language and the approved annexes. Individual projects can introduce additional annexes to support project-specific analysis needs. Use of such annexes results in AADL models that are not fully compliant with the standard. Standard compliant AADL tools are however required to accept such AADL specifications (see *Processing Requirements and Permissions).*

Examples of annex libraries are constraint functions that can be referenced in assertions.

Syntax

```
annex_subclause ::=
     annex annex_identifier  {**
        annex_specific_language_constructs
     **} ;
```

```
annex_library ::=
    **annex** *annex*_identifier  {**
       annex_specific_reusable_constructs
    **} ;
```

Naming Rules

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

Legality Rules

Annex subclauses can only be declared in component types and component implementations.

A component type or component implementation declaration may contain at most one annex subclause for each annex.

Annex libraries can only be declared in packages.

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

Semantics

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.

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 predicate function expressed in a constraint language such as the Object Constraint Language (OCL) notation.

Processing Requirements and Permissions

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

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 annexes. Processing methods compliant with a given annex must process specifications as defined in that annex.

Annex-specific 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 OCL {**
   pre: 0 < Input_Sample < maxValue;
   post: 0 < Output_Sample < maxValue;
**};
end Collect_Samples;
```

5 Software Components

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

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.

Data components represent data types and static data in source text. Data components are sharable between threads within the same thread group or process, and across processes and systems.

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

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 server subprogram calls, and through shared data components.

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

A process represents a virtual address space. Access protection of the virtual address space is enforced at runtime 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

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, is represented by data subcomponents in a data component implementation. Subprogram features of a data component type can model the concept of methods on a class or operations on an abstract data type. If subprogram features are declared, the data component may only be accessed through the subprograms. 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.

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.

A data subcomponent represents static data in the source text. Components can have shared access to data subcomponents. Only those components that explicitly declare required data access can access such sharable data subcomponents according to a specified concurrency control protocol property. Concurrency control is assured either through the subprogram features of the data component type or by the component requiring access. Data subcomponents can be shared within the same process and, if supported by the runtime system, across processes.

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.

Legality Rules

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

A data type declaration can contain subprogram declarations, provides data access declarations as well as property associations.

A data type declaration must not contain a flow specification.

A data implementation can contain data subcomponents, a modes subclause, access connections, and data property associations.

A data implementation must not contain a flow implementation or an end-to-end flow specification.

Each requires data access reference may have its own `Required_Access` property value. This property value must not conflict with the `Provided_Access` property value associated with the data component or the corresponding provides access declaration.

The data classifier references of two data ports, event data ports, data access, or parameters to be identical. When a data type has zero or one implementation, then the referenced data types must match. When a data type has more than one implementation, both the data type and the appropriate implementation must be present in the data classifier reference of a data port, event data port, data access, or parameter declaration.

Standard Properties

```
Source_Data_Size: Size

Type_Source_Name: aadlstring

Source_Name: aadlstring

Source_Text: inherit list of aadlstring

-- Data sharing properties

Concurrency_Control_Protocol: Supported_Concurrency_Control_Protocols =>
                                                            NoneSpecified
```

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 data variable in the source text.

Semantics

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 subprogram declarations in the **features** subclause of the data type declaration. In this case, the data may be accessed through the subprograms.

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.

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.

A data component type declaration can provide access to its data subcomponents. This allows other components to directly access specific parts of the data component represented by the data subcomponent for which access is provided. This can be used to model source language concepts such as direct access to fields of a record or public access to instance variables in a class.

A data component type declaration can require access to data components external to the data component type and its implementations. This can be used to model references to other data in the source language.

Data component types can be extended through component type extension declarations. This permits modeling of subclasses and type inheritance in source text.

A data subcomponent represents a data instance, i.e., static data in the source text that is potentially sharable between threads and persists across thread dispatches. Each declared data subcomponent represents a separate instance of source text data.

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 even generate a system instance and perform memory usage analysis if a `Source_Data_Size` property value is specified in the data component type.

Static data is sharable 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 `Required_Access` property value. Its value must be consistent with the value of the `Provided_Access` property.

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. Upon exit from a critical region a `Release_Resource` operation is performed. If multiple data components with concurrency control protocols are accessed by a thread, the critical regions may be nested, i.e., the `Get_Resource` and `Release_Resource` operations are pair-wise nested for each data component. Furthermore, deadlock avoidance among threads accessing the same set of shared data components is assured by proper nesting of the critical regions across all of the threads.

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. Data component classifier references are considered to be compatible if they are identical, and if the represented source text data types are compatible according to source language rules.

Data implementation property associations allow mode-specific property values to be associated with the data component.

NOTES:

The property types **aadlboolean**, **aadlstring**, **aadlinteger**, and **aadlreal** cannot be used as predeclared component data types. Instead data component types with the names Boolean, string, integer, and real can be declared in a package and used throughout AADL specifications.

Processing Requirements and Permissions

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.

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.

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.

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.

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.

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

```
data Person
end Person;

data Personnel_record
-- Methods are not required, but when provided act as access methods
features
   -- a subprogram feature with reference to a
   -- subprogram type for signature
   update_address: subprogram 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.
   Name : data basic::string;
   Home_address : data sei::aadl::relief::Address;
end Personnel_record.others;

subprogram update_address
features
   person: in out parameter Personnel_record;
   street :in parameter basic::string;
   city: in parameter basic::string;
end update_address;

package basic
public
   -- string as type
   data string
   end string;

   -- int as type
   data int
   properties
      Source_Data_Size => 64 bits;
   end int;
end basic;

-- 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;

package sei::aadl::relief
public
   data Address
   features
      -- a subprogram feature without parameter detail
      getStreet : subprogram;
      getCity : subprogram;
   end Address;
end sei::aadl::relief;

-- The implementation is shown as a private declaration
-- The public and the private part of a package are separate AADL spec's
package sei::aadl::relief
private
   data implementation Address.others
   subcomponents
      street : data basic::string;
      streetnumber: data basic::int;
```

```
        city: data basic::string;
        zipcode: data basic::int;
    end Address.others;
end sei::aadl::relief;
```

5.2 Subprograms and Subprogram Calls

A subprogram component represents an execution entrypoint in source text. A subprogram may not have any internal state (static data). 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.

Subprograms can be called from threads and from other subprograms. These calls are sequential calls local to the virtual address space of the thread. Subprograms can also be called remotely from threads in other virtual address spaces through server subprograms (see Section 8.3). A subprogram call sequence is declared in a subprogram or thread implementation and may be mode-specific.

Syntax

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

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

called_subprogram ::=
        subprogram_classifier_reference
        | data_unique_type_reference . data_subprogram_identifier
```

NOTES:

Subprogram type and implementation declarations follow the syntax rules for component types and implementations. Subprograms are not instantiated as subcomponents. Instead subprogram calls represent their instantiation (use) with a specific set of parameter values. The syntax for specifying call sequences is shown here.

Naming Rules

The defining identifier of a subprogram call sequence declaration, if present in a component implementation, must be unique within the local namespace of the component implementation that contains the subprogram call.

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

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

If the called subprogram name is a subprogram feature reference in a data component, the data component type identifier must refer to a data component type, or it must refer to a requires data access declaration in the component type of the component containing the subprogram call declaration.

Legality Rules

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

A subprogram type declaration can contain parameter, out event port, out event data port, and port group declarations as well as required data access declarations. It can also contain a flow specification subclause as well as property associations.

A subprogram implementation can contain a connections subclause, a subprogram calls subclause, a flows subclause, a modes subclause and subprogram property associations.

A subprogram implementation must not contain a subcomponent subclause.

Only one subprogram call sequence can apply to a given mode. In other words, a mode identifier can be specified in the `in_modes` subclause of at most one subprogram call sequence.

Standard Properties

```
-- Properties related to source text
Source_Name: aadlstring
Source_Text: inherit list of aadlstring
Source_Language: Supported_Source_Languages
-- 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)
-- execution related properties
```

```
Compute_Execution_Time: Time_Range
Compute_Deadline: Time
-- remote subprogram call related properties
Actual_Subprogram_Call: reference (server subprogram)
Allowed_Subprogram_Call: list of reference (server subprogram)
Actual_Subprogram_Call_Binding: reference (bus, processor, memory)
Allowed_Subprogram_Call_Binding:
   inherit list of reference (bus, processor, device)
Queue_Processing_Protocol: Supported_Queue_Processing_Protocols => FIFO
```

Semantics

A subprogram component represents sequentially executable source text that is called with parameters. 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 In addition, events being raised from within the subprogram through its **out event port** and **out event data port**, and required access to static data by the subprogram are specified as part of the features subclause of a subprogram type declaration.

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.

If required data access is declared for a subprogram type, access to the data subcomponent is performed in a critical region to assure concurrency control for calls from different threads (for more on concurrency control see Sections 5.1 and 5.3).

Subprogram source text can contain `Raise_Event` service calls to cause the transmission of events and event data through its **out event** ports. 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.

Subprogram implementations and thread implementations can contain subprogram calls. 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 9.1.2).

A thread or subprogram can contain multiple calls to the same subprogram - with the same parameters or with different parameters.

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

Declaration of subprogram calls in thread and subprogram implementations implies that an instance of the subprogram executable binaries exists in the load image of the process that contains the thread performing the subprogram calls. For subprograms, whose source text implementation is reentrant, it is assumed that a single instance of the subprogram binaries exist in the process virtual 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.

Subprogram calls can be calls to server subprograms provided in other threads. Such calls model synchronous remote subprogram calls. An `Allowed_Subprogram_Call` property, if present, identifies the server subprogram(s) that are allowed to be used in a call binding. An `Actual_Call_Binding` property records the actual binding to a server subprogram. Constraints on the buses and processors over which such calls can be routed can be specified with the `Allowed_Subprogram_Call_Bindings` property.

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.

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.

Processing Requirements and Permissions

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.

Examples

```
data Matrix
end Matrix;

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: subprogram getCurrent;
   getFuture: subprogram getFuture;
end weather_DB;
```

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

subprogram getFuture
-- a subprogram whose source text contains a raise_event service call
-- 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 subprogram 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 subprogram 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;
   event 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 Threads

A thread represents a sequential flow of control that executes instructions within a binary image produced from source text. A thread models a schedulable unit that transitions between various scheduling states. 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.

Systems modeled in AADL can have operational modes (see Section 11). 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, or at a predeclared in event port called `Dispatch`. Completion of the execution of a thread dispatch will result in an event being delivered through the predeclared `Complete` event out port if it is connected.

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 propagated as an event through the predeclared `Error` out event data port.

Legality Rules

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

A thread type declaration can contain port, port group, server subprogram declarations as well as requires and provides data access declarations. It can also contain a flow specifications and property associations.

A thread component implementation can contain data declarations, a calls subclause, a flows subclause, a modes subclause, and thread property associations.

A thread or any of its features may not contain an explicit `Dispatch` **in** event or event data port declaration, nor a `Complete` or `Error` **out** event or event data port declaration.

The `Compute_Entrypoint` property must have a value that indicates the source code to execute after a thread has been dispatched. Other entrypoint properties are optional, i.e., if a property value is not defined, then the entrypoint is not called.

The `Period` property must have a value if the `Dispatch_Protocol` property value is `periodic` or `sporadic`.

Standard Properties

```
-- Properties related to source text
Source_Text: inherit list of aadlstring
Source_Language: 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
Period: inherit Time
-- the default value of the deadline is that of the period
Deadline: Time => inherit value(Period)
-- Properties specifying execution entrypoints and timing constraints
```

```
Initialize_Execution_Time: Time_Range
Initialize_Deadline: Time
Initialize_Entrypoint: aadlstring
Compute_Execution_Time: Time_Range
Compute_Deadline: Time
Compute_Entrypoint: aadlstring
Activate_Execution_Time: Time_Range
Activate_Deadline: Time
Activate_Entrypoint: aadlstring
Deactivate_Execution_Time: Time_Range
Deactivate_Deadline: Time
Deactivate_Entrypoint: aadlstring
Recover_Execution_Time: Time_Range
Recover_Deadline: Time
Recover_Entrypoint: aadlstring
Finalize_Execution_Time: Time_Range
Finalize_Deadline: Time
Finalize_Entrypoint: aadlstring
-- Properties specifying constraints for processor and memory binding
Allowed_Processor_Binding_Class:
   inherit list of classifier (processor, system)
Allowed_Processor_Binding: inherit list of reference (processor, system)
Allowed_Memory_Binding_Class:
   inherit list of classifier (memory, system, processor)
Allowed_Memory_Binding: inherit list of reference (memory, system, processor)
Not_Collocated: list of reference (data, thread, process, system, connections)
Allowed_Connection_Binding_Class:
   inherit list of classifier(processor, bus, device)
Allowed_Connection_Binding: inherit list of reference (bus, processor, device)
Actual_Connection_Binding: inherit reference (bus, processor, device)
-- Binding value filled in by binding tool
Actual_Processor_Binding: inherit reference (processor)
Actual_Memory_Binding: inherit reference (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
         => value(Default_Active_Thread_Handling_Protocol)
Active_Thread_Queue_Handling_Protocol:
   inherit enumeration (flush, hold) => flush
```

Semantics

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 5.5).

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.

Threads can contain mode subclauses that define thread-internal operational modes. Threads can have property values that are different for different thread-internal modes.

Every thread has a predeclared **in event port** named `Dispatch`. If this port is connected, i.e., named as the destination in a connection declaration, then the arrival of an event results in the dispatch of the thread. If the `Dispatch` port is connected, then the arrival of an event on an explicitly declared event ports or event data ports will result in the queuing of the event or event data without causing a thread dispatch. When the `Dispatch` port is connected, only events arriving at this port will cause a thread to be dispatched.

Periodic threads ignore any events arriving through explicitly declared event or event data connections or through an event connection to the `Dispatch` port. Periodic thread dispatches are solely determined by the clock according to the time interval specified through the `Period` property value.

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 the execution of a thread dispatch completes.

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. This supports the propagation of thread unrecoverable errors as event data for fault handling by a thread. The source text implementing the fault handling thread may map the error represented by event data into an event that can trigger a mode switch through a `Raise_Event` call in its source text.

NOTES:

Mode transitions can only refer to event ports as their trigger. This means that `Error` ports cannot be directly connected to mode transitions. 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.

Thread States and Actions

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 5.5).

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 the **Execution Fault Handling** section below.

The initial state is *thread halted*. When the loading of the virtual address space as declared by the enclosing process completes (see Section 5.5), 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.

When a mode switch 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 18). If the thread is periodic and its `Synchronized_Component` property is true, then its period is taken into consideration to determine the actual mode switch time (see Sections 11 and 12.3 for detailed timing semantics of a mode switch). If an aperiodic or a sporadic thread is executing a dispatch when the mode switch is initiated, its execution is handled according to the `Active_Thread_Handling_Protocol` property. 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*.

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. 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 8.1 and 9.1.1). 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.

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 5.5), then its virtual address space is reloaded, the processor to which the thread is bound is restarted, or the system instance is restarted.

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.

Figure 5 presents the top-level hybrid automaton (using the notation defined in Section 1.6) to describe the dynamic semantics of a thread. Two succeeding figures elaborate on the `Performing` substate (Figure 6 and 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) and with system wide mode switching (see Section 12.1). 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.

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 **Runtime Support**). The execution semantics for these entrypoints is described in the **Thread Scheduling and Execution** section that follows.

An *initialize entrypoint* is executed once during system initialization and allows threads to perform application specific initialization, such as insuring the correct initial value of its **out** and **in out** ports.

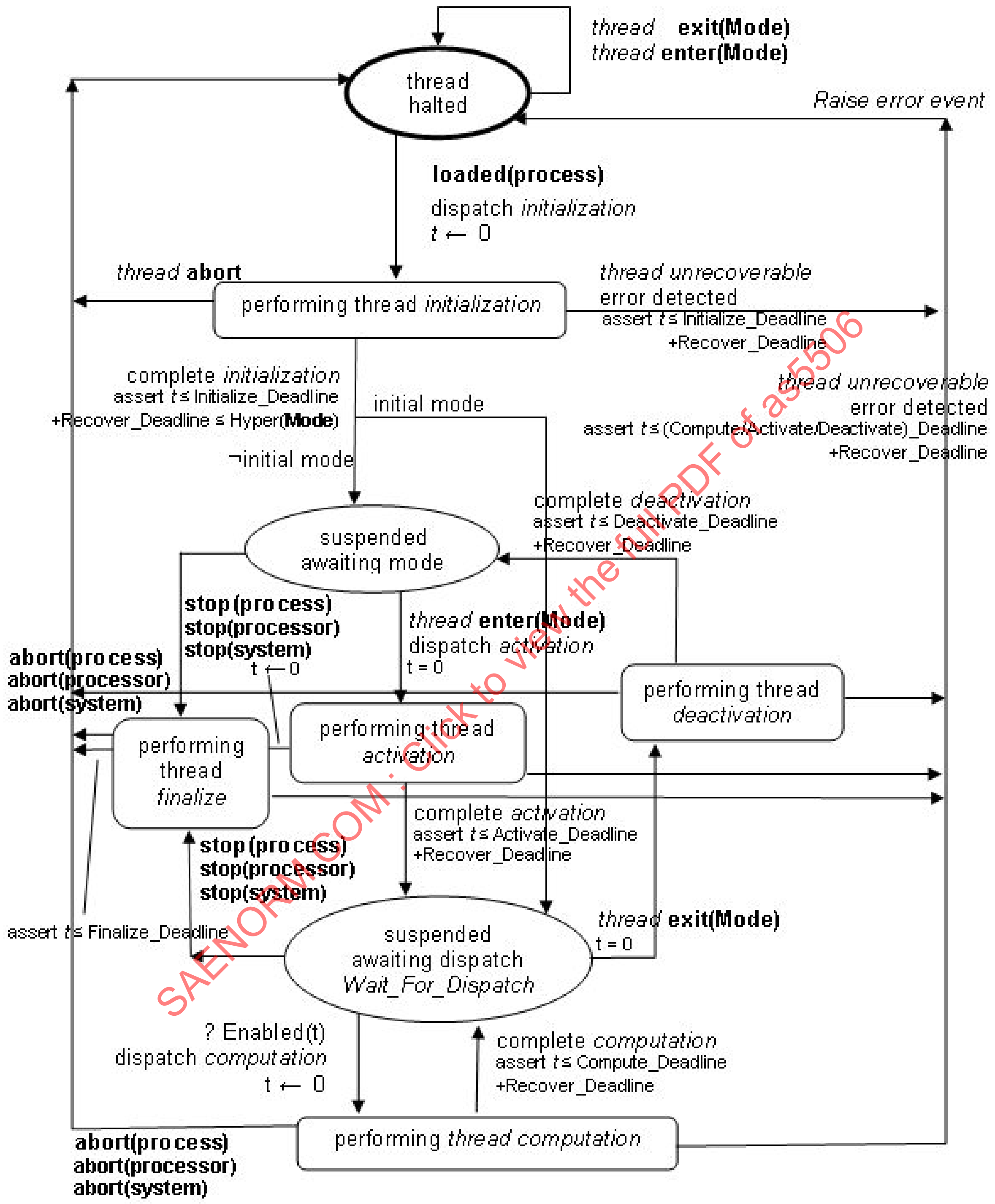
The *activate* and *deactivate entrypoints* are executed during mode transitions and allow threads to take user-specified actions to save and restore application state for continued execution between mode switches.These entrypoints may be used to reinitialize application state due to a mode transition. Activate entrypoints can also ensure that **out** and **in out** ports contain correct values for operation in the new mode.

The *compute entrypoint* represents the code sequence to be executed on every thread dispatch. Each server subprogram represents a separate compute entrypoint of the thread. Server 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.

A *recover entrypoint* 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 the **Execution Fault Handling** section below).

A *finalize entrypoint* is executed when a thread is asked to terminate as part of a process unload or process stop.

If no value is specified for any of the entrypoints, execution is considered as immediately completed without consumption of execution time.



thread halted
thread **exit(Mode)**
thread **enter(Mode)**
Raise error event
loaded(process)
dispatch *initialization*
t ← 0
thread **abort**
performing thread *initialization*
thread unrecoverable error detected
assert *t* ≤ Initialize_Deadline +Recover_Deadline
complete *initialization*
assert *t* ≤ Initialize_Deadline +Recover_Deadline ≤ Hyper(**Mode**)
initial mode
¬initial mode
thread unrecoverable error detected
assert *t* ≤ (Compute/Activate/Deactivate)_Deadline +Recover_Deadline
complete *deactivation*
assert *t* ≤ Deactivate_Deadline +Recover_Deadline
suspended awaiting mode
stop(process)
stop(processor)
stop(system)
t ← 0
thread **enter(Mode)**
dispatch *activation*
t = 0
abort(process)
abort(processor)
abort(system)
performing thread *deactivation*
performing thread *finalize*
performing thread *activation*
complete *activation*
assert *t* ≤ Activate_Deadline +Recover_Deadline
stop(process)
stop(processor)
stop(system)
assert *t* ≤ Finalize_Deadline
thread **exit(Mode)**
t = 0
suspended awaiting dispatch *Wait_For_Dispatch*
? Enabled(t)
dispatch *computation*
t ← 0
complete *computation*
assert *t* ≤ Compute_Deadline +Recover_Deadline
performing *thread computation*
abort(process)
abort(processor)
abort(system)

Figure 5 Thread States and Actions

Thread Dispatching

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 and the Wait_For_Dispatch invariant. The Enabled function determines when a transition 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 **Thread Scheduling and Execution** below).

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 $t \le Period$. The dispatch occurs at $t = Period$.

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 server subprogram feature of the thread. The time interval between successive dispatch requests will never be less than the associated Period property value. The Overflow_Handling_Protocol property for event ports specifies the action to take when events arrive too frequently, i.e., the time between successive events is less than what is specified in the associated Period property. These events are either ignored, queued until the end of the period (and then dispatched), or are treated as an error. The Enabled function is $t \ge Period$. The Wait_For_Dispatch invariant is true. The dispatch actually occurs when the guard 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 guard becomes true, or a remote subprogram call arrives when the guard is true.

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 server subprogram feature of the thread. There is no constraint on the inter-arrival time of events, event data or remote subprogram calls. The Enabled function is true. The Wait_For_Dispatch invariant is true. 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.

If several events or event data occur logically simultaneously and are routed to the same port of an aperiodic or sporadic 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 or sporadic thread, the order of event handling is determined by the Urgency property associated with the ports.

For a thread whose dispatch protocol is background, the thread is dispatched immediately upon completion of its initialization entrypoint execution. The Enabled function is true. The Wait_For_Dispatch invariant is $t = 0$. The dispatch occurs immediately. 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. 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.

Thread Scheduling and Execution

When a thread action is *computation*, 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 *execute nominally* or *execute 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).

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.

Resource blocking can occur when two threads attempt to access shared data. Such access is performed in a critical region. When a thread enters a critical region a `Get_Resource` operation is performed on the shared data component. Upon exit from a critical region a `Release_Resource` operation is performed. 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).

A running thread may require access to shared resources such as shared data components through a critical region. Such access is coordinated through a concurrency control mechanism that implements the specified concurrency control protocol. These mechanisms may be blocking such as the use of a semaphore or non-blocking such as non-preemption through priority ceiling protocol. In the case of a blocking mechanism, a thread entering a critical region (via `Get_Resource`) may be blocked and enter the *awaiting resource* state. The thread transitions out of the awaiting resource state into the ready state when another thread exits the critical region (via `Release_Resource`). Multiple threads may block trying to gain access to the same resource; such access is mediated by the specified concurrency coordination protocol (see Section 5.1).

The time a thread resides in a critical region, i.e., the time it may block other threads from entering the critical region, in worst case is the duration of executing one thread dispatch. This time may be reduced by specifying a call sequence within a thread and indicating the subprogram(s) that require access to shared data, i.e., have to execute in a critical region.

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.

Subprogram calls to server subprograms are synchronous. A thread in the running state enters the *awaiting return* state when performing a call to a subprogram whose service is performed by a server subprogram in another thread. The service request for the execution of the subprogram is transferred to the server subprogram request queue of a thread as specified by the `Actual_Subprogram_Call` property that specifies the binding of the subprogram call to a server subprogram feature in another thread. When the thread executing the corresponding server subprogram completes and the result is available to the caller, the thread with the calling subprogram transitions to the ready state.

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 resume* state. If the thread was in a critical region, it will be suspended once it exits 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 resume state into the ready state. A background thread must be allowed to exit any critical region before it can be suspended as result of a mode switch.

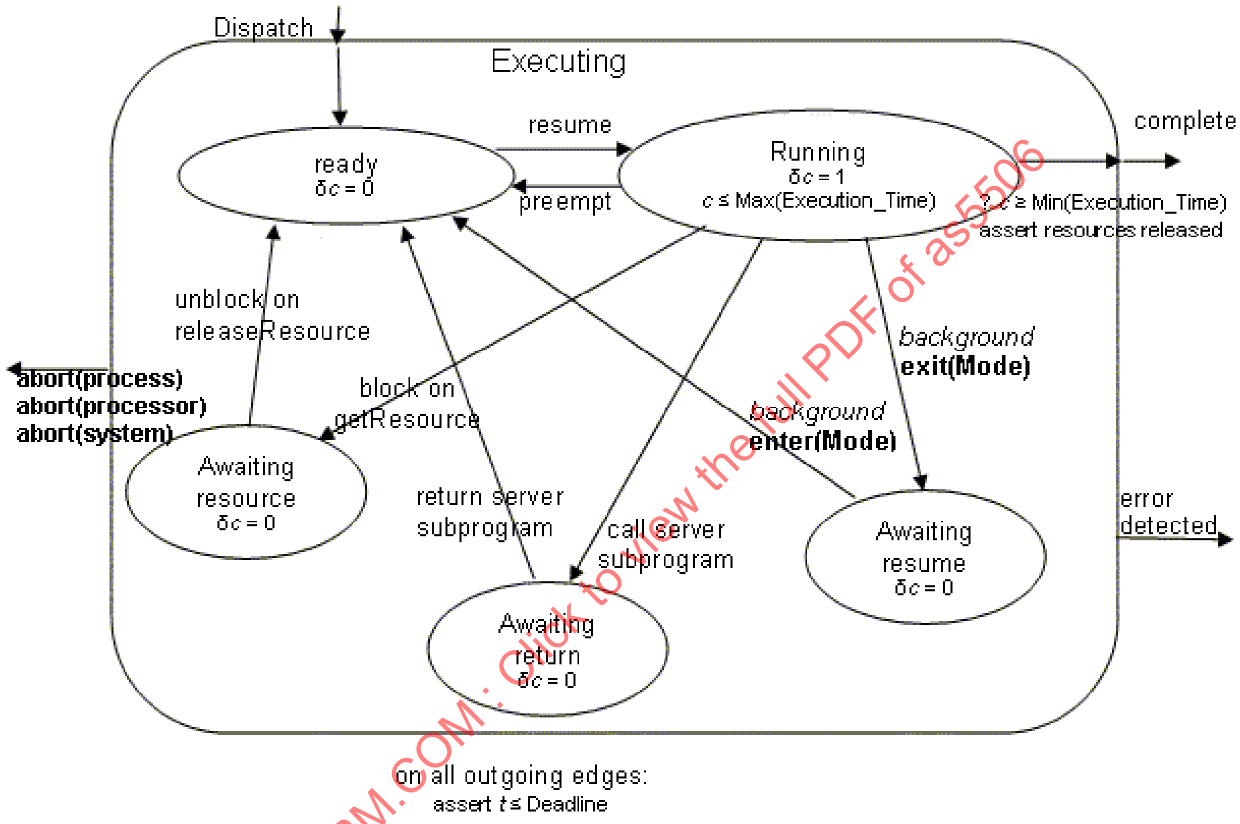


Figure 6 Thread Scheduling and Execution States

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 δ*c* and δ*t*. A derivative value of 1 indicates time accumulation and a value of 0 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.

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.

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. An implementation must document its handling of timing constraints.

Execution Fault Handling

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 operation and is not handled as part of normal execution by that thread.

Detectable errors may be classified as *thread recoverable* errors, or *thread unrecoverable* errors. In the case of a thread recoverable error, the thread can recover and continue with the next dispatch. Thread unrecoverable errors can be communicated as events and handled as thread dispatches or mode changes. Alternatively, these errors may be reported as event data and communicated to an error handling thread for further analysis and recovery actions.

For thread recoverable errors, 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. 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. This thread-level fault handling behavior is illustrated in Figure 7.

In the presence of a thread recoverable error, the maximum interval of time between the dispatch of a thread and its returning to the suspensed 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.

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 suspensed awaiting dispatch state is the compute deadline, and the maximum execution time consumed is the compute execution time.

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.

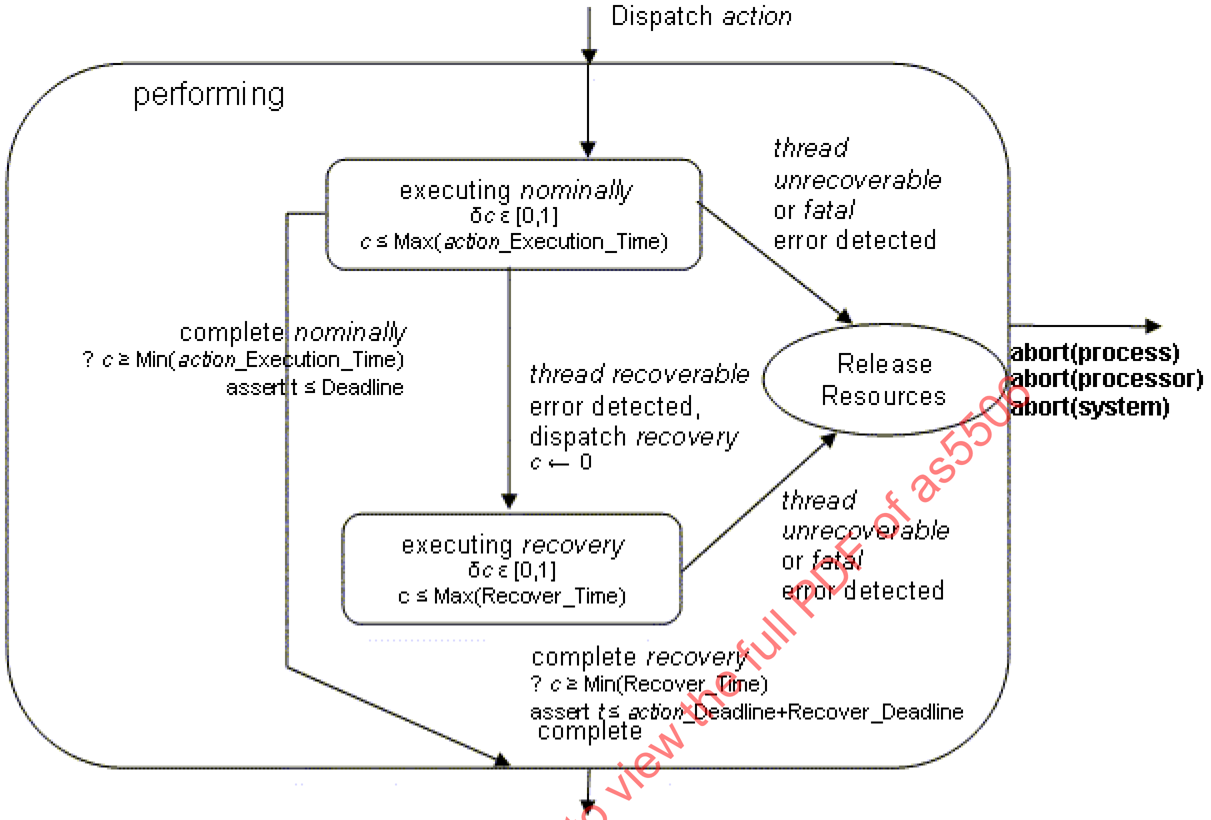


Figure 7 Performing Thread Execution with Recovery

A fault may be detected by the source text runtime system or by the application itself. Detection of a fault in the source text runtime system can result in an exception that is caught and handled within the source text. The source text exception handler may propagate the error to an external handler by raising an event or an event with data.

For errors encountered by the source text runtime system, the error class is defined by the developer.

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.

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

If a fault is encountered by the application source text 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 recovery entrypoints, the error class must be *thread unrecoverable*.

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.

System Synchronization Requirements

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 global real time.

Two periodic threads are said to be synchronized if, whenever they are both active in the current system mode of operation, they are logically dispatched simultaneously at every global real time *ST* that is a nonnegative integral multiple of the least common multiple of their periods, i.e., their hyperperiod. Two threads are logically dispatched simultaneously 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. If all periodic threads contained in an application system are synchronized, then that application system is said to be synchronized. In this version of the standard, system instances are synchronized. By default, all application system instances are synchronized, i.e., all periodic threads contained in an application system must be synchronized.

Runtime Support

The following paragraphs define standard runtime services that are to be 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.

A `Raise_Event` runtime service shall be provided that allows a thread to explicitly raise an event if the executing thread has the named port specified as out event port or an out event data port.

A `Raise_Error` runtime service shall be provided that allows a thread to explicitly raise a thread recoverable or thread unrecoverable error as specified by a runtime parameter.

If a local subprogram calls `Raise_Event`, the event is routed according to the event connection declaration associated with the subprogram call's event port. If a server subprogram calls `Raise_Event`, the event is transferred to the subprogram call and routed according to the event connection declaration associated with the subprogram call's event port.

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 server subprogram, the error is passed to the error port of the thread executing the server subprogram.

Processing Requirements and Permissions

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.

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.

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.

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.

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.

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.

A method of implementing the `Raise_Event` service call may provide an optional parameter that permits the assignment of an `Urgency` value to the event. Such an `Urgency` value provides control over the implementation-dependent ordering of events or event data. This is the case for logically simultaneous events or event data arriving at the same event or event data port, and for logically simultaneous events resulting in different mode transitions. This capability also allows priority-based `Queue_Processing_Protocols` to be supported for event and event data ports.

A method of implementing the `Raise_Event` service call may provide a status return value indicating whether the raised event or event data connected to an event or event data port is ignored on delivery to an event or event data port, or whether a raised event that triggers a mode transition is ignored.

A method of implementation may choose to generate the `Get_Resource` and `Release_Resource` calls in support of critical regions for shared data access 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.

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.

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.

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.

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.

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.

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.

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.

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 point prior to the initialization deadline for that periodic thread.

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 server subprogram calls made by that thread. The deadline property of a server subprogram may be useful for scheduling methods that assign intermediate deadlines in the course of producing an overall end-to-end system schedule.

5.4 Thread Groups

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.

A thread group does not represent a virtual address space nor does it represent a unit of execution. Therefore, a thread group must be contained within a process.

Legality Rules

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

A thread group component type can contain provides and requires data access, as well as port, port group, and server subprogram declarations. It can also contain flow specifications and property associations.

A thread group component implementation can contain data, thread, and thread group declarations.

An instantiable thread group component implementation must contain at least one thread subcomponent or one thread group subcomponent.

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

A thread group must not contain a subprogram calls subclause.

Standard Properties

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

Active_Thread_Handling_Protocol:
   inherit Supported_Active_Thread_Handling_Protocols
         => value(Default_Active_Thread_Handling_Protocol)
Period: inherit Time
Deadline: Time => inherit value(Period)
-- Properties specifying constraints for processor and memory binding
Allowed_Processor_Binding_Class:
   inherit list of classifier (processor, system)
Allowed_Processor_Binding: inherit list of reference (processor, system)
Actual_Processor_Binding: inherit reference (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 reference (memory)
Allowed_Connection_Binding_Class:
   inherit list of classifier(processor, bus, device)
Allowed_Connection_Binding: inherit list of reference (bus, processor, device)
Actual_Connection_Binding: inherit reference (bus, processor, device)
```

NOTES:

Property associations of thread groups are inheritable (see Section 10.3) 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

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.

A thread group implementation contains threads, data components, 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.

5.5 Processes

A process represents a virtual address space. The `Runtime_Protection` process property indicates whether this virtual address space is runtime protected, i.e., it represents a space partition unit whose boundaries are enforced at runtime. The virtual address space contains the program formed by the source text associated with the process and its subcomponents. A complete implemenation of a process must contain at least one thread or thread group subcomponent.

Legality Rules

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

A process component type can contain port, port group, provides and requires data access, and server subprogram declarations. It can also contain flow specifications and property associations.

A process component implementation can contain data, thread, and thread group declarations.

A complete process component implementation must contain at least one thread subcomponent or one thread group subcomponent.

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

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 wth the data and subprograms that are referenced by required subcomponent declarations.

Standard Properties

```
-- Runtime enforcement of virtual address space boundary
Scheduling_Protocol: list of Supported_Scheduling_Protocols
-- Properties related to source text
Source_Text: inherit list of aadlstring
Source_Language: Supported_Source_Languages
-- Properties related to virtual address space loading
Load_Time: Time_Range
Load_Deadline: Time
-- Inhertable thread properties
Synchronized_Component: inherit aadlboolean => true

Active_Thread_Handling_Protocol:
   inherit Supported_Active_Thread_Handling_Protocols
         => value(Default_Active_Thread_Handling_Protocol)
Period: inherit Time
Deadline: Time => inherit value(Period)
-- Properties specifying constraints memory binding
Allowed_Processor_Binding_Class:
   inherit list of classifier (processor, system)
Allowed_Processor_Binding: inherit list of reference (processor, system)
Actual_Processor_Binding: inherit reference (processor)
Allowed_Connection_Binding_Class:
   inherit list of classifier(processor, bus, device)
Allowed_Connection_Binding: inherit list of reference (bus, processor, device)
Actual_Connection_Binding: inherit reference (bus, processor, device)
```

```
Allowed_Memory_Binding_Class:
   inherit list of classifier (memory, system, processor)
Allowed_Memory_Binding: inherit list of reference (memory, system, processor)
Not_Collocated: list of reference (data, thread, process, system, connections)
Actual_Memory_Binding: inherit reference (memory)
```

Semantics

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.

Threads contained in a process execute within the virtual address space of the process. If the `Runtime_Protection` property value is true, the virtual address space boundaries of the process are enforced for all contained threads at runtime.

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 events.

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.

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.

The process states, transitions, and actions are illustrated in Figure 8. Once processors of an execution platform are started, binary images making up the virtual address space of processes bound to the processor are ready to be loaded. 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).

A process, i.e., its contained threads, can be stopped (also known as a deferred abort), which is indicated by **stop(process)**. 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. When a process is stopped, each of its threads is given a chance to execute its finalize entrypoint.

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).

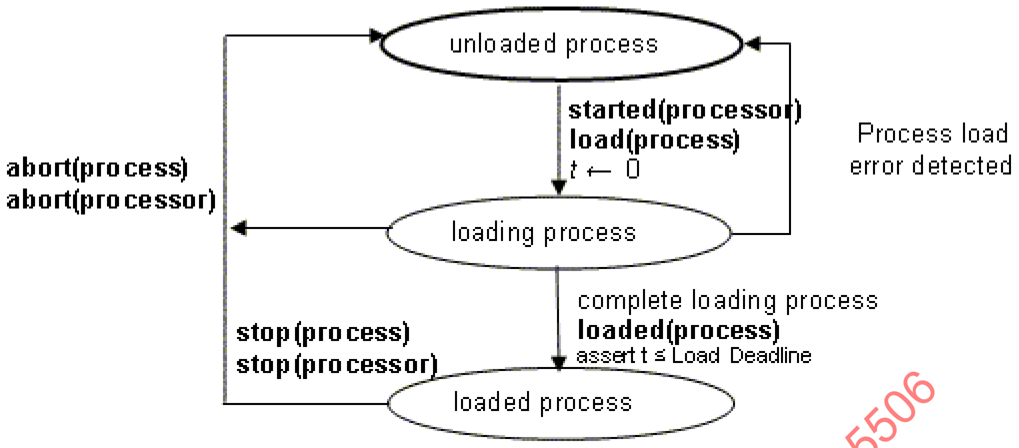


Figure 8 Process States and Actions

Processing Requirements and Permissions

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.

If the applicable source language standard permits a mixture of source languages, then subcomponents may have different source language property values.

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.

5.6 Predeclared Runtime Services

Language-specific annexes to this standard will define predeclared subprograms that are included with every thread, and process implementation. Names used for explicitly declared components, features, connections and behaviors must be distinct from the names of all standard predeclared components.

Every language-specific annex will define a source language application program interface for the predeclared subprograms, shown below.

The following subprograms can be called by any application thread from its source text:

```
Raise_Event : subprogram ;
Raise_Error : subprogram ;
```

The following are subprograms, whose calls are only necessary in application source text, if the particular implementation method does not perform those functions as part of its AADL runtime support.

```
Await_Dispatch : subprogram ;
Get_Resource : subprogram ;
Release_Resource : subprogram ;
```

6 Execution Platform Components

This section describes the four categories of execution platform components: processor, device, memory and bus.

Processors can execute threads. Processors can contain memory subcomponents. Processors and devices can access memories over buses.

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

Buses provide access between processors, devices, and memories. A bus provides the resources necessary to perform exchanges of control and data as specified by connections. 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.

Devices represent entities that interface with the external environment of an application system and may have complex behaviors. A device can interact with application software components through their port and subprogram features. 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.

Processors may contain software subcomponents, where the associated source text and data in the form of binary images will be bound to memories accessible from that processor. These software components implement the capability of the processor to schedule and execute threads bound to that processor.

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 modeler.

The hardware represented by the execution platform components may be modeled by a hardware description or simulation language. Alternatively, they may be represented using configuration data for programmable logic devices. Or a simulation may be used to characterize the components. Such descriptions are also viewed as associated source text.

Execution platform components can be used to model a layered system architecture. Processors, buses, memory, and devices may represent abstractions of a virtual machine layer. Those abstractions can be modeled as systems in terms of software components and of execution platform components of the next virtual machine layer; eventually a system architecture layer representing physical hardware. The mapping between different layers of a multi-layer architecture may be represented by an appropriate property for each of the execution platform categories. For example, a `Maps_To` property may be defined to specify the mapping of an execution platform component classifier to a system classifier that represents the implementation of the abstraction in the virtual machine layer.

6.1 Processors

A processor is an abstraction of hardware and software that is responsible for scheduling and executing threads. Processors execute threads declared in application software systems and in devices that can be accessed from those processors. Processors may contain memories and may access memories and devices via buses.

Legality Rules

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

A processor component type can contain port, port group, server subprogram, and required bus access declarations. It may contain flow specifications as well as property associations.

A processor component implementation can contain declarations of memory subcomponents.

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

A processor implementation must not contain a connection subclause or a subprogram calls subclause.

A processor component must contain at least one memory component or require at least one bus access.

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: 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 reference (memory)
-- Processor initialization properties
Startup_Deadline: inherit Time
-- Properties specifying provided thread execution support
```

```
Thread_Limit: aadlinteger 0 .. value(Max_Thread_Limit)
                                             => value(Max_Thread_Limit)
Allowed_Dispatch_Protocol: list of Supported_Dispatch_Protocols
Allowed_Period: list of Time_Range
Server_Subprogram_Call_Binding: inherit list of reference (thread, processor)
Process_Swap_Execution_Time: Time_Range
Thread_Swap_Execution_Time: Time_Range
Supported_Source_Language: list of Supported_Source_Languages
-- Proeprties related to data movement in memory
Assign_Time: Time
Assign_Byte_Time: Time
Assign_Fixed_Time: Time
-- Properties related to the hardware clock
Clock_Jitter: Time
Clock_Period: Time
Clock_Period_Range: Time_Range
```

NOTES:

The above is 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

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.

A processor to which threads are bound must have a `Scheduling_Protocol` property value.

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.

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.

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

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

The source text property may include 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.

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.

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 17) 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. Once a processor is initialized it is operational and ready to load virtual address spaces of the processes whose threads are bound to the processor. 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 software. After process virtual address spaces are loaded, process initialization entrypoints are executed, if they have been specified. At this point, the **started(system)** and **started(processor)** transitions have completed and the processor is in the *processor operating* state.

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

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 5.3 and 5.5. A stop processor request results in a transition to the processor stopping state at the next hyperperiod. 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 11). When the next hyperperiod begins, the 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(process)** in the hybrid automata figures in Section 5.3. In this case, any threads bound to the processor are permitted to complete their dispatch executions and perform any finalization before the processor is stopped.

The synchronization scope for **start(processor)** and **stop(processor)** in Figure 9 consists of all processes whose threads that are bound to the processor. The edge labels **start(processor)** and **stop(processor)** also appear in the set of concurrent semantic automata for processes.

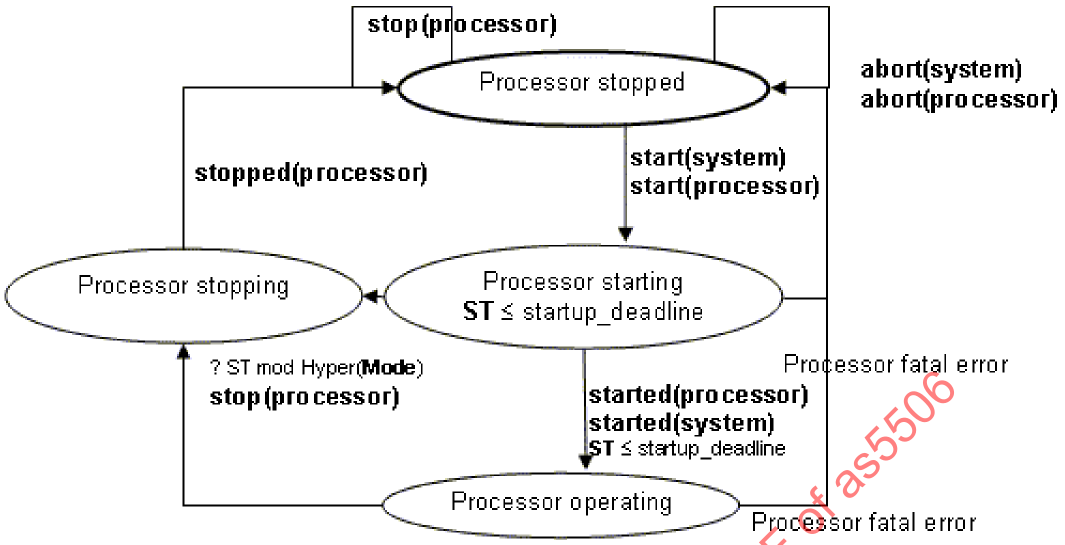


Figure 9 Processor States and Actions

Processing Requirements and Permissions

A method of implementation is not required to monitor the startup deadline and report an overflow as an error.

6.2 Memory

A memory component represents an execution platform component that stores binary images. 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 Flow specifications: no Properties: yes	Subcomponents: • memory Subprogram calls: no Connections: no Flows: no Modes: yes Properties: yes

A memory type can contain requires bus access declarations and property associations. It must not contain flow specifications.

A memory implementation can contain memory subcomponent declarations.

A memory implementation can contain a modes subclause and property associations.

A memory implementation must not contain a connections subclause, 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
Word_Count: aadlinteger 0 .. value(Max_Word_Count)
Word_Space: aadlinteger 1 .. value(Max_Word_Space) => 1
Base_Address: access aadlinteger 0 .. value(Max_Base_Address)
Read_Time: list of Time_Range
Write_Time: list of Time_Range
-- Hardware description properties
Hardware_Description_Source_Text: inherit list of aadlstring
Hardware_Source_Language: Supported_Hardware_Source_Languages
```

Semantics

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.

A memory is accessible from a processor if the memory is connected via a shared bus component and the `Allowed_Access_Protocol` property value for that bus includes `Memory_Access`.

Memory components can have different property values under different operational modes.

6.3 Buses

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.

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.

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.

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 Flow specifications: no Properties: yes	Subcomponents: • None Subprogram calls: no Connections: no Flows: no Modes: yes Properties: yes

A bus type can have requires bus access declarations and property associations.

A bus type must not contain any flow specifications.

A bus implementation must not contain any subcomponent declarations.

A bus implementation can contain a modes subclause and property associations.

A bus implementation must not contain a connections subclause, flows subclause, or subprogram calls subclause.

Standard Properties

```
-- Properties specifying bus transmission properties
Allowed_Connection_Protocol: list of enumeration
                              (Data_Connection,
                               Event_Connection)
Allowed_Access_Protocol: list of enumeration (Memory_Access,
                                              Device_Access)
Allowed_Message_Size: Size_Range
Transmission_Time: list of Time_Range
Propagation_Delay: Time_Range
-- Hardware description properties
Hardware_Description_Source_Text: inherit list of aadlstring
Hardware_Source_Language: Supported_Hardware_Source_Languages
-- Data movement related properties
Assign_Time: Time
Assign_Byte_Time: Time
Assign_Fixed_Time: Time
```

Semantics

A bus provides access 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_Protocol` 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.

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 and the `Allowed_Connection_Protocol` property value for that bus includes `Device_Access`. A memory is accessible from a processor if the two share a bus component and the `Allowed_Connection_Protocol` property value for that bus includes `Memory_Access`.

Buses can be directly connected to other buses. This is represented by one bus declaration specifying access to another bus in its requires subclause.

Bus components can have different property values under different operational modes.

Processing Requirements and Permissions

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.

A method of implementation shall define the methods used for bus arbitration and scheduling.

Examples

```
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;
```

6.4 Devices

A device component represents an execution platform component that interfaces with the external environment. A device may exhibit complex behavior that requires a nontrivial interface to application software systems. Devices may internally have a processor, memory and software that is not explicitly modeled. 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.

A device interacts with both execution platform components and application software components. A device has 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.

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.

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.

Examples of devices are sensors and actuators that interface with the external physical world, or standalone systems (such as a GPS) that interface with an application system.

Legality Rules

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

A device type can contain port, port group, server subprogram, requires bus access declarations, flow specifications, as well as property associations.

A device component implementation must not contain a subcomponents subclause, connections subclause, or subprogram calls subclause.

A device implementation can contain 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: Supported_Source_Languages
Source_Code_Size: Size
Source_Data_Size: Size
Source_Stack_Size: Size
-- Properties specifying the thread properties for device software
-- executing on a processor
Device_Dispatch_Protocol: Supported_Dispatch_Protocols => Aperiodic
Period: inherit Time
Compute_Execution_Time: Time_Range
Deadline: Time => inherit value(Period)
-- Properties specifying constraints for processor and memory binding
Allowed_Memory_Binding_Class:
   inherit list of classifier (memory, system, processor)
Allowed_Memory_Binding: inherit list of reference (memory, system, processor)
Allowed_Processor_Binding_Class:
    inherit list of classifier (processor, system)
Allowed_Processor_Binding: inherit list of reference (processor, system)
```

Semantics

A device component represents an execution platform component that provides an interface with the external environment. It can exhibit complex behaviors that require a nontrivial interface to application software systems via ports and subprogram features. This functionality may be fully embedded in the device hardware, or it may be provided by device-specific software. 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.

A device is accessible from a processor if the device is connected via a shared bus component and the `Allowed_Connection_Protocol` property value for that bus includes `Device_Access`.

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.

Device components can have different property values under different operational modes.

Processing Requirements and Permissions

Execution of the device driver software may be considered to be part of the processor execution overhead or it may be treated as an explicitly declared thread with its own execution properties.

7 System Composition

Systems are organized into a hierarchy of components to reflect the structure of physical 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

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. This provides for modeling of large-scale runtime architectures.

Legality Rules

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

A system component type can contain provided and required data and bus access declarations, port, port group, and server subprogram declarations. It can also contain flow specifications as well as property associations.

A system component implementation can contain data, process, and system subcomponent declarations as well as execution platform components, i.e., processor, memory, bus, and device.

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: Supported_Source_Languages
-- Process property that can be specified at the system level as well
-- Runtime enforcement of address space boundaries
Scheduling_Protocol: list of Supported_Scheduling_Protocols
-- Inhertable thread properties
Synchronized_Component: inherit aadlboolean => true
Active_Thread_Handling_Protocol:
   inherit Supported_Active_Thread_Handling_Protocols
         => value(Default_Active_Thread_Handling_Protocol)
Period: inherit Time
Deadline: Time => inherit value(Period)
-- Properties related binding of software component source text in
-- systems to processors and memory
Allowed_Processor_Binding_Class:
   inherit list of classifier (processor, system)
Allowed_Processor_Binding: inherit list of reference (processor, system)
Actual_Processor_Binding: inherit reference (processor)
Allowed_Memory_Binding_Class:
   inherit list of classifier (memory, system, processor)
Allowed_Memory_Binding: inherit list of reference (memory, system, processor)
Not_Collocated: list of reference (data, thread, process, system, connections)
Actual_Memory_Binding: inherit reference (memory)
Allowed_Connection_Binding_Class:
   inherit list of classifier(processor, bus, device)
Allowed_Connection_Binding: inherit list of reference (bus, processor, device)
Actual_Connection_Binding: inherit reference (bus, processor, device)
-- Properties related systems as execution platforms
Available_Processor_Binding: inherit list of reference (processor, system)
Available_Memory_Binding: inherit list of reference (memory, system)
Hardware_Source_Language: Supported_Hardware_Source_Languages
-- Properties related to startup of processor contained in a system
Startup_Deadline: inherit Time
-- 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

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.

System components can represent application software components and connections that must be bound to execution platform components, i.e., processors, memories, and buses in order to be executable. Those software components and connections may be bound to execution platform components within the system, or they must be bound to execution platform components outside the system.

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 execution platform components. They represent aggregations of execution platform components that act as the execution platform.

Some system components are self-contained in that all software components and connections are bound to execution platform components contained within the 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 can also be modeled as devices. The device representation takes a black-box perspective, while the system representation takes a white-box perspective.

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 the system and is triggered by the arrival of events. A system can have mode-specific property associations.

8 Features and Shared Access

A *feature* is a part of a component type definition that specifies how that component interfaces with other components in the system. The four categories of features are: port, subprogram, parameters, and subcomponent access.

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. *Port groups* represent groups of component ports or port groups. Port groups can be used anywhere ports can be used. Within a component, the ports of a port group can be connected to individually. Outside a component, port groups can be connected as a single unit.

The *subprogram* feature represents a call interface for a service that is accessible to other components. *Server subprogram* features represent subprograms that execute in their own thread and can be called remotely. Data subprogram features represent subprograms through which the data component is manipulated. Call sequences (see Section 5.2) specify calls to subprogram classifiers, data subprogram features, and server subprogram features.

Parameter features represent data values that can be passed into and out of subprograms. Parameters are typed with a data classifier reference.

Subcomponent access represents communication via shared access to data and bus components. A data or 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 data or bus subcomponent declared outside utilizing a requires access feature declaration.

Syntax

```
feature ::=
    port_spec |
    port_group_spec |
    server_subprogram_spec |
    data_subprogram_spec |
    subcomponent_access |
    parameter

feature_refinement ::=
    port_refinement |
    port_group_refinement |
    server_subprogram_refinement |
    data_subprogram_refinement |
    subcomponent_access_refinement |
    parameter_refinement
```

Naming Rules

The defining identifier of a feature must be unique within the interface namespace of the associated component type.

Thread features may not be declared using the predeclared ports names `Dispatch`, `Complete` or `Error`.

Each refining feature identifier that appears in a feature refinement declaration must also appear in a feature declaration in the associated component type or one of its ancestors.

A feature is named 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

The feature classifier reference and the port direction in a refined feature declaration must be identical to the feature classifier reference and the port direction in the refined declaration.

In the case of data and event data ports, the refined feature declaration in a component type extension can complete an incomplete data classifier reference.

Feature refinements can associate new property values.

Each feature can be refined at most once in the same component implementation or type extension.

Semantics

A feature declaration specifies an externally accessible element of a component. Features are also visible from within component implementations associated with the component type that contains the feature declaration.

A refined feature declaration may complete an incomplete component classifier reference and declare feature property associations.

8.1 Ports

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 look like data components, i.e., they are data variables accessible in the source text. From the perspective of the application source text, event ports represent `Raise_Event` runtime service calls and transfer the event to receiving components or to the system executive to trigger a mode change. Event data ports transfer the data along with the event to receiving components.

Syntax

```
port_spec ::=
    defining_port_identifier : ( in | out | in out ) port_type
        [ { { port_property_association }+ } ] ;
```

```
port_refinement ::=
    defining_port_identifier : refined to
        ( in | out | in out ) port_type
        [ { { port_property_association }+ } ] ;


port_type ::=
     data port [ data_classifier_reference ]
    |  event data port [ data_classifier_reference ]
    |  event port
```

Naming Rules

A defining port identifier adheres to the naming rules specified for all features (see Section 8).

The unique component type identifier 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.

Legality Rules

Ports can be declared in subprogram, thread, thread group, process, system, processor, and device component types.

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.

Data 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.

The property names `Overflow_Handling_Protocol`, `Queue_Processing_Protocol`, `Dequeue_Protocol` and `Queue_Size` may only appear in property associations for **in** event ports and **in** event data ports.

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
-- Optional property for device ports
Device_Register_Address: aadlinteger
```

```
-- Port specific compute entrypoint properties for event and event data ports
Compute_Entrypoint: aadlstring
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)
-- In port queue properties
Queue_Size: aadlinteger 0 .. value(Max_Queue_Size) => 0
Queue_Processing_Protocol: Supported_Queue_Processing_Protocols => FIFO
Overflow_Handling_Protocol: enumeration (DropOldest, DropNewest, Error)
                                                                => DropOldest
Urgency: aadlinteger 0 .. value(Max_Urgency)
Dequeue_Protocol: enumeration ( OneItem, AllItems ) => OneItem
```

Semantics

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.

A data or event data port represents a data instance that maps to a static variable in the source text. This mapping is 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.

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_Time` and `Compute_Deadline` takes precedence over those of the containing thread.

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 single static 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.

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.

Data and event data ports are used to transmit data between threads. They appear to the thread as input and output buffers, accessible in source text as port variables. In the case of a data out port, data is automatically transmitted at the completion of thread dispatch execution. In the case of an event data out port, data is automatically transmitted at the time of the `Raise_Event` runtime service call (see Section 9).

Data ports are intended for transmission of state data such as signals. 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, which is accessible through the port variable.

Event ports are used to communicate events. Events can be raised by source text executing in subprograms, threads, and by processors and devices. Events can trigger the dispatch of other threads or can cause a mode switch. If the receiving thread is already active, the event is queued. Events are triggered by an explicit `Raise_Event` runtime service call executed within a thread. Or if the thread's predeclared `Complete` port (see Section 5.3) is connected, then a runtime system completion event is generated.

Event ports are represented by port variables. When a thread dispatch is the result of an event, the value of the event port variable is set to one and the event is dequeued. If the `Dequeue_Protocol` property is set to `AllItems,` then the value of the event port variable is set to the number of events in the queue and the event queue is emptied. The port variable value of other event ports is set to zero.

When the thread dispatch is triggered through the predeclared `Dispatch` port or, in the case of periodic threads, by the clock, then the port variable value of all event ports is set to one and one event is dequeued. If the `Dequeue_Protocol` property is set to `AllItems`, then the value of each port variable is set to the number of queued events at that time and the queue is emptied. This capability allows a periodic thread to sample an event stream to determine the number of events that arrive in a given time interval. For example, it permits the thread to determine the speed of a wheel based on a rotational sensor on the wheel. It also permits a system health monitor thread to periodically process system alarms without overloading the processor due to spikes in alarm arrivals.

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. If the receiving thread is not executing a dispatch and the `Dispatch` port is not connected, then the arrival of a message triggers a dispatch of the receiving thread. The message transmission is triggered by an explicit `Raise_Event` runtime service call on the specific event data port. If not transmitted through an explicit `Raise_Event` runtime service call, then the event data is transmitted at completion of dispatch execution for those event data ports into which new values were written.

Event data ports are represented by port variables. The status of whether an event data port provides a new value to a thread is accessible through the port variable. If the queue is empty at the time of thread dispatch, the event data port variable retains its old value.

When a thread dispatch is the result of an event, the value of the event data port variable is set to the data value of the event data in the queue that triggered the dispatch and the event data is dequeued. If the `Dequeue_Protocol` property is set to `AllItems,` then the data of all event data is placed in the port variable and the event data queue is emptied. The port variables of other event data ports retain their old value. If the thread dispatch is triggered through the predeclared `Dispatch` port, then the port variable value for each event data ports is set to the data value of the first event data in the queue and the event data is dequeued. If the `Dequeue_Protocol` property is set to `AllItems,` then all event data is placed in the port variable for each event data port and the event data queue is emptied.

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 `Raise_Event` runtime service calls. It may also raise an event at completion through its predeclared `Complete` port (see Section 5.3) and transmit event data through event data ports that contain new values that have not been transmitted through explicit `Raise_Event` runtime service calls.

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.

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 11 and 12.3).

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.

Queues will be serviced in a first-in, first-out order. When an event-driven thread declares multiple in event and event data ports in its type 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 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 .

Processing Requirements and Permissions

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.

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.

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.

A method of implementing systems must provide a capability for a thread to determine whether a data port has been updated with a new value since the previous dispatch. This capability may be implemented in the form of a "fresh" field in the port variable. A runtime service call may be provided to supply port variable status information.

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.

A method of processing specifications is permitted to use non-standard property names and associations to define alternative queuing disciplines.

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 B; end GPS;
   data INS properties Source_data_Size => 20 B; end INS;
   data Position_ECEF properties Source_data_Size => 30 B; end Position_ECEF;
   data Position_NED properties Source_data_Size => 30 B; end Position_NED;
end Nav_Types;

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;
end Blended_Navigation;

process implementation Blended_Navigation.Simple
subcomponents
   Integrate : thread;
   Navigate : thread;
```

```
end Blended_Navigation.Simple;
```

8.2 Port Groups and Port Group Types

Port groups represent groups of component ports or port groups. Within a component, the ports of a port group can be connected to individually. Outside a component, port 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 ports from one subcomponent and its contained subcomponents must be connected to ports in another subcomponent and its contained subcomponents. The content of a port group is declared through a port group type declaration. This declaration is then referenced when port groups are declared as component features.

Syntax

```
-- Defining the content structure of a port group
port_group_type ::=
    port group defining_identifier
    ( features
       { port_spec | port_group_spec }*
      [ inverse of unique_port_group_type_reference ]
    |
      inverse of unique_port_group_type_reference
    )
 [ properties ( { portgroup_property_association }+ | none_statement ) ]
 { annex_subclause }*
end defining_identifier ;

port_group_type_extension ::=
    port group defining_identifier extends unique_port_group_type_reference
    ( features
        { port_spec | port_refinement |
          port_group_spec | port_group_refinement }*
      [ inverse of unique_port_group_type_reference ]
    |
      inverse of unique_port_group_type_reference
    )
 [ properties ( { portgroup_property_association }+ | none_statement ) ]
 { annex_subclause }*
end defining_identifier ;
```

```
-- declaring a port group as component feature
port_group_spec ::=
  defining_port_group_identifier : port group
      [ unique_port_group_type_reference ]
        [ { { portgroup_property_association }+ } ] ;

port_group_refinement ::=
  defining_port_group_identifier : refined to
      port group [ unique_port_group_type_reference ]
        [ { { portgroup_property_association }+ } ] ;

unique_port_group_type_reference ::=
    [ package_name :: ] port_group_type_identifier
```

Naming Rules

The defining identifier of a port group type must be unique within the package namespace of the package where the port group type is declared. If the port group type is declared in the AADL specification directly, it must be unique within the anonymous namespace.

Each port group provides a local namespace. The defining port identifiers of port and port group declarations in a port group type must be unique within the namespace of the port group type. The local namespace of a port group type extension includes the defining identifiers in the local namespace of the port group type being extended. This means, the defining identifiers of port or port group declarations in a port group type extension must not exist in the local namespace of the port group type being extended. The defining identifiers of port or port group refinements in a port group type extension must refer to a port or port group in the local namespace of an ancestor port group type.

The defining port identifiers of `port_spec` declarations in port group refinements must not exist in the local namespace of any port group being extended. The defining port identifier of `port_refinement` declarations in port group refinements must exist in the local namespace of any port group being extended.

The package name of the unique port group type reference must refer to a package name in the global namespace. The port group type identifier of the unique port group type reference must refer to a port group type identifier in the named package namespace, or if the package name is absent, in the anonymous namespace.

Legality Rules

A port group type may contain zero or more elements, i.e., ports or port groups. If it contains zero elements, then the port group type may be declared to be the inverse of another port group type. Otherwise, it is considered to be incompletely specified.

A port group type can be declared to be the inverse of another port group type, as indicated by the reserved words **inverse of** and the name of a port group type. Any port group type named in an **inverse**

of statement cannot itself contain an **inverse of** statement. This means that several port groups can be declared to be the inverse of one port group. However, chaining of inverses such as B inverse of A and C inverse of B is not permitted.

A port group type that is an extension of another port group type cannot contain an **inverse of** statement. The port group type being extended cannot contain an **inverse of** statement.

Two port group types are considered to complement each other if the following holds:

- The number of ports or port groups contained in the port group and its complement must be identical.
- Each of the declared ports or port groups in a port group must be a pair-wise complement with that in the port group complement, with pairs determined by declaration order.

If both port group types have zero features, then they are considered to complement each other. In the case of port group type extensions, the port and port group declarations in the extension are considered to be after the declarations in the port group type being extended. Ports are pair-wise complementary if they have complementary direction (**out** / **in**, **in** / **out**, **in out** / **in out**), and are of the same port type. In the case of event data or data ports, the data component classifier reference must be identical.

A port group declaration that does not specify a port group type reference is incomplete. Such a reference can be added in a port group refinement declaration.

A port group declaration may be refined by adding a property association. Inclusion of the port group type reference is optional.

If the `Aggregate_Data_Port` property of a port group has the value true, all ports contained in its port group type or the port group type of any contained port group must be data ports and they must all have the same port direction.

Standard Properties

```
Aggregate_Data_Port: aadlboolean =>  false
-- Port properties defined to be inherit, thus can be associated with a
-- port 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)
```

Semantics

A port group declaration represents groups of component ports that can be connected to externally through a single connection. Port groups can contain port groups. This supports nested grouping of ports for different levels of the modeled system.

Within a component, the ports of a port group can be connected to individually. The members of the port group are declared in a port group type declaration that is referenced by the port group declaration. The referenced port group type determines the port group compatibility for a port group connection.

The **inverse of** reserved words of a port group type declaration indicate that the port group type represents the complement to the referenced port group type. The legality of port group connections is affected by the complementary nature of port groups (see Section 9).

The AADL supports the concept of *aggregate data port* as an extension of the port group concept. A port group property called `Aggregate_Data_Port` identifies the role of the port group as an aggregate data port. This port property applies to all **out** data ports and **in out** data ports of the port group.

The role of an aggregate data port is to make a collection of data from multiple **out** 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.

Processing Requirements and Permissions

The functionality of an aggregate data port can be realized as a thread whose only role is to collect the data values from several **in** data ports and make them available at the same time on their respective **out** data ports. 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.

Examples

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


port 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;


port group MyGPS_plug
   -- second port group as inverse of the original
   -- no chaining in inverse and
   -- no pairwise inverse references are allowed
   inverse of GPSbasic_socket
end MyGPS_plug;
```

```
port group GPSextended_plug extends GPSbasic_plug
features
   Signal: out event port;
   Cmd: in data port GPSLib::commands;
end GPSextended_plug;

process Satellite_position
features
   position:   port group GPSBasic_socket;
end Satellite_position;

process GPS_System
features
   position: port group GPSbasic_plug;
end GPS_System;

system implementation Satellite.others
subcomponents
   SatPos: process Satellite_position;
   MyGPS: process GPS_System;
connections
   port group Satpos.position -> MyGPS.position;
end Satellite.others;
```

8.3 Subprograms As Features

A data subprogram feature represents an execution entrypoint in source text that operates on a data component of the associated data type. Server subprogram features represent entrypoints for remote procedure calls, i.e., the ability to synchronously call this subprogram from a separate thread that may execute on a different processor.

Syntax

```
data_subprogram_spec ::=
  defining_subprogram_identifier : subprogram
      [ subprogram_classifier_reference ]
      [ { { subprogram_property_association }+ } ] ;
```

```
data_subprogram_refinement ::=
  defining_subprogram_identifier : refined to subprogram
      [subprogram_classifier_reference ]
      [ { { subprogram_property_association }+ } ] ;

server_subprogram_spec ::=
  defining_subprogram_identifier : server subprogram
       [ unique_subprogram_reference ]
        [ { { subprogram_property_association }+ } ] ;

server_subprogram_refinement ::=
  defining_subprogram_identifier : refined to
    server subprogram
       [ unique_subprogram_reference ]
        [ { { subprogram_property_association }+ } ] ;

unique_subprogram_reference ::=
    subprogram_classifier_reference
    | data_subprogram_feature_classifier_reference

data_subprogram_feature_classifier_reference ::=
    [ package_name :: ] data_type_identifier . subprogram_identifier
```

Naming Rules

A unique subprogram reference must be a subprogram classifier reference or a reference to a subprogram feature declaration in a data component type.

Legality Rules

Data subprogram features can be declared in data components and must not have the reserved word **server**.

Server subprogram features can be declared in thread, thread group, process, processor, and system component types. They must have the reserved word **server**.

A server subprogram feature declaration must only refer to a subprogram classifier or to a subprogram feature in a data component type.

If several subprogram declarations refer to the same subprogram type or via the `Source_Name` property to the same subprogram in the source text, then their parameter signatures in the source text and the property associations must be consistent with each other.

When a server subprogram declaration appears in a thread component type, the scheduling protocol property value for all thread implementations, subcomponents, and instances having that component type must be `Aperiodic` or `Sporadic`.

A subprogram refinement can specify a subprogram type reference and declare property associations.

For calls to server subprograms, the subprogram classifier or subprogram feature reference of the subprogram call and the subprogram classifier or subprogram feature reference of the server subprogram must be the same.

Standard Properties

```
Source_Name: aadlstring
Source_Text: inherit list of aadlstring
Source_Stack_Size: Size
-- Subprogram execution properties
Subprogram_Execution_Time: Time_Range
-- Client subprogram execution properties
Client_Subprogram_Execution_Time: Time
-- Server subprogram execution properties
Compute_Execution_Time: Time_Range
Compute_Deadline: Time
Recover_Execution_Time: Time_Range
Recover_Deadline: Time
Overflow_Handling_Protocol: enumeration (DropOldest, DropNewest, Error)
                                                   => DropOldest
Queue_Size: aadlinteger 0 .. value(Max_Queue_Size) => 0
```

Semantics

Data subprogram features represent entrypoints into source text that operate on data components of the associated data component type. They are called by naming a data component type and the subprogram separated with a '.' (dot) (see also Section 5.2).

Data subprogram features can refer to separately declared subprogram classifiers, which may specify the parameters, required access, and out event or out event data ports of the subprogram (see Section 5.2).

If server subprogram features refer to subprograms in source text with parameters, the parameters are marshalled and unmarshalled as necessary.

If server subprogram features refer to subprograms that raise events or event data, then the raised event in the server subprogram is mapped to the corresponding event or event data port in the caller subprogram.

Threads and subprograms can contain subprogram calls (see Sections 5.2 and 5.3). These can be calls to subprograms local to thread, or they can be synchronous remote calls to a server subprogram in another thread that is indicated by the `Actual_Subprogram_Call` property. In case of a remote call, the requesting thread calls a local proxy that carries out the service request. The execution time of the client proxy is determined by the `Client_Subprogram_Execution_Time`. The actual call results in communicating the subprogram execution request. 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.

Server subprogram features model service requests such as remote procedure calls to services provided by a thread. Actual calls are specified as explicit subprogram calls whose call binding property specifies a server subprogram.

Server subprogram features can be declared with their signature defined in the referenced and separately declared subprogram classifiers.

A server subprogram feature declaration in a thread component type represents an entrypoint to a remotely callable code sequence in the source text associated with a different thread. This thread may reside in the same process, a different process on the same processor, or on a different processor. A request for execution of such a subprogram is a dispatch request to the thread containing the server subprogram – the same way events represent dispatch requests for aperiodic or sporadic threads. As in the case of events, requests for execution of server subprograms may be queued if the thread is already executing a dispatch request. A thread can have multiple subprogram entrypoints, expressed by multiple server subprogram feature declarations in the respective component type. Only one of these server subprograms may be executed per thread dispatch. Queuing and queue servicing follows the semantics of event port queues.

If an event is raised during the execution of a server subprogram, this event is communicated back to the calling subprogram and propagated according to the connection declaration associated with the event port of the caller (see also Section 5.3).

If an error occurs or is raised while the thread is executing the server subprogram, that error is communicated through the `Error` port of the containing thread - as previously defined in Section 5.3.

Processing Requirements and Permissions

Every method for processing specifications must parse subprogram feature 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 subprogram features. In this case, subprogram features may be rejected as unsupported. In this case, a warning message may be generated to notify the user of this behavior in the toolset.

A subprogram feature of a software component maps to a subprogram declared in associated source text as defined in the source language annex of this standard. This source subprogram must be declared and visible at the outermost lexical scope as defined by the applicable source language standard. The parameter identifiers appearing in the subprogram feature declaration must map to formal parameter names appearing in the source subprogram declaration. Rules for mapping subprogram and parameter identifiers to source text are defined in the source language annex of this standard. Any special rules for server subprograms and parameter are defined in the source language annex of this standard.

A call to a server subprogram feature of a software component maps to a call to the proxy for the remote subprogram in the associated source text. This proxy routine performs the appropriate remote invocation. Actual calls to the proxy subprogram in the source text are found in the source program of the calling thread. The data types of the actual parameter expressions must be compatible with the declared data types in the specification as defined by the applicable source language standard. The details of these mappings are defined in the source language annex of this standard.

A method of implementation may automatically generate the source text required to perform a remote subprogram call. This may include the marshalling and unmarshalling parameter values or the transmission and reception of call and return events.

If the calling thread can execute on the same processor as the server subprogram, then a method of implementation may use the calling thread to execute the server subprogram code rather than a separate thread. When using this optimization procedure, proper synchronization must be maintained to preserve the same runtime semantics. If runtime address space protection is required, then this technique cannot be used if the protection between the calling thread and server subprogram thread is lost as result of the optimization.

NOTES:

The annex subclause may be used along with the existing syntax to build a component-based client/server subprogram model. An example of this approach follows.

Examples

```
package Pierre
public
   process printers
   features
      printonServer : server subprogram print;
      mainPrinter: in event port;
      backupPrinter: in event port;
   end printers;

   process implementation printers
   subcomponents
      A : thread printer in modes ( modeA );
      B : thread printer in modes ( modeB );
   Modes
      modeA: initial mode;
      modeB: mode;
      modeA -[ backupPrinter ]-> modeB;
      modeB -[ mainPrinter ]-> modeA;
   end printers;

   thread printer
   features
      print : server subprogram print;
   end printer;
   subprogram print
   features
      filetoprint: data file;
   end print;

end Pierre;

thread A
features
   print: requires subprogram Pierre::print;
calls
   print: subprogram;
end A;

-- example of a server subprogram call
process B
annex pierre {**
-- a new features declaration to indicate a required subprogram
features
   remoteprint: requires subprogram Pierre::print;
**}
end B;

-- example of a local subprogram call
-- the subprogram is locally declared within the process.
-- the call is bound to it.
process C
end C;

process implementation C.default
subcomponents
   app: thread A;
annex pierre {**
-- a new subcomponents declaration to specify a local subprogram
subcomponents
```

```
   localprint: subprogram Pierre::print;
   -- a new connection declaration to bind a required subprogram to a
   -- local subprogram or a server subprogram via connections
connections
   subprogram app.print -> localprint;
**}
end C.default;
```

8.4 Subprogram Parameters

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_classifier_reference ]
        [ { { parameter_property_association }+ } ] ;

parameter_refinement ::=
    defining_parameter_identifier : refined to
        ( in | out | in out ) parameter [ data_classifier_reference ]
        [ { { parameter_property_association }+ } ] ;
```

Naming Rules

A defining parameter identifier adheres to the naming rules specified for all features (see Section 8).

The data classifier reference must refer to a data component type or a data component implementation.

Legality Rules

Parameters can be declared for subprogram component types.

If a parameter refinement includes a data classifier reference, then the classifier reference must be the same as that of the parameter being refined.

If the parameter being refined has an incomplete data classifier reference, then the parameter refinement may complete an incompletely specified data classifier reference.

A parameter refinement cannot redefine the direction of a parameter.

Standard Properties

```
-- Properties specifying the source text representation of the parameter
Source_Name: aadlstring
Source_Text: inherit list of aadlstring
```

Semantics

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.

8.5 Subcomponent Access

Subcomponents can be made accessible outside their containment hierarchy. Components can declare that they require access to externally declared subcomponents. Components may provide access to their subcomponents. Provided subcomponent access is a feature of a component.

A *required subcomponent access* declaration in the component type of the subcomponent indicates that a subcomponent requires access to a data component declared external to the component. Required subcomponent accesses are resolved to actual data subcomponents as part of a subcomponent declaration. Different forms of required access, such as read-only access, are specified by a `Required_Access` property.

A *provides subcomponent access* declaration in the component type of the subcomponent indicates that a subcomponent provides access to a data component contained in the component. Provided subcomponent accesses can be used to resolve required subcomponent access. Different forms of provided access, such as read-only access, are specified by a `Provided_Access` property.

A subcomponent that is accessed by more than one subcomponent is shared. The actual (shared) subcomponent may be declared within the same component implementation as the one(s) requiring access or it may be declared higher in the component containment hierarchy. Alternatively, it may be declared within a subcomponent at the current level (of the ones requiring access) or higher. In this case, the containing subcomponent will specify that it provides access to the shared subcomponent.

This is illustrated in Figure 10. Data D is a data component contained in the process implementation of process subcomponent A. The process type of A makes it accessible through its **provides data access** feature declaration. It is being accessed by thread Q, which is contained as subcomponent of the process implementation for process B. Both the process type of process B and the thread type of thread Q indicate the need to access a data component through a **requires data access** feature declaration. In the system implementation of system Simple the **provides data access** feature of process A is connected to the **requires data access** feature of process B through a **data access** connection. The textual AADL model of this specification is given as an example later in this section.

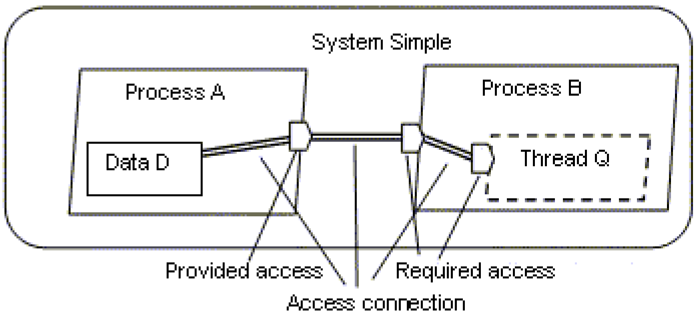


Figure 10 Containment Hierarchy and Shared Access

Syntax

```
-- The requires and provides subcomponent access subclause
subcomponent_access ::=
    defining_subcomponent_access_identifier :
       subcomponent_access_classifier
        [ { { access_property_association }+  } ] ;

subcomponent_access_refinement ::=
    defining_subcomponent_access_identifier : refined to
        subcomponent_access_classifier
        [ { { access_property_association }+  } ] ;

subcomponent_access_classifier ::=
        ( provides |  requires ) ( data | bus ) access
        [ unique_component_type_identifier
                         [ . component_implementation_name ] ]
```

Naming Rules

The defining identifier of a provides or requires subcomponent access declaration must be unique within the interface namespace of the component type where the subcomponent access is declared.

The defining identifier of a provides or requires subcomponent refinement must exist as a defining identifier of a required subcomponent in the interface namespace of the associated component type or one of its ancestors.

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

Legality Rules

The category of the subcomponent access declaration must be identical to the category of the component type (and of the component implementation) in the referenced subcomponent classifier.

Standard Properties

```
Required_Access : access enumeration (read_only, write_only, read_write,
                                      by_method) => read_write

Provided_Access : access enumeration (read_only, write_only, read_write,
                                      by_method) => read_write
```

Semantics

The requires subcomponent access declaration indicates that the component requires access to a subcomponent not contained in any of the implementations of the component type with the requires subcomponent access declaration. The `Required_Access` property specifies how a component of a given component type accesses a required subcomponent component that may be shared by multiple subcomponents. The reference to a required subcomponent is resolved, i.e., bound to a subcomponent, when a subcomponent of the component type requiring access is declared. When required subcomponent references of two different subcomponents are bound to the same subcomponent, the subcomponent is shared by them.

The provided subcomponent access declaration indicates that a subcomponent contained in the component implementations is made accessible outside the component. The `Provided_Access` property indicates how the shared data component may be accessed.

Examples

```
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;
```

9 Connections and Flows

A *connection* is a linkage 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.

A *flow* is a logical flow of information through a sequence of threads, processors, devices, and 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.

9.1 Connections

The AADL supports three types of connections: *port connections*, *parameter connections*, and *access connections*. Port connections represent the transfer of data and control between two concurrently executing components, i.e., between two threads or between a thread and a processor or device. Parameter connections denote the flow of data through the parameters of a sequence of subprogram calls, i.e., between units of sequential execution within a thread. Access connections designate access to shared data components by concurrently executing threads or by subprograms executing within a thread. They also represent communication between processors, memory, and devices by accessing a shared bus.

Syntax

```
connection ::=
    port_connection
    | parameter_connection
    | access_connection

connection_refinement ::=
    port_connection_refinement
    | parameter_connection_refinement
    | access_connection_refinement
```

9.1.1 Port Connections

Port connections represent transfer of data and control between two concurrently executing components, i.e., between two threads or between a thread and a processor or device. 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*. An individual port connection declaration links a port of one subcomponent to the port of another. Or it joins a port of a subcomponent with a port of a containing component.

Semantic port connections are illustrated in Figure 11. The *ultimate source* of a semantic port connection is an outgoing feature, i.e., an **out** or **in out** port of a thread, processor, or device component. The *ultimate destination* of a semantic port connection is an incoming feature, i.e., an **in** or **in out** port of a thread subcomponent, a processor or device component. In the case of event connections and port group connections, a mode transition may also be specified as part of the port connection.

Port connection declarations follow the containment hierarchy of threads, thread groups, processes and systems. Some connections link an outgoing feature to the corresponding feature in the containing component and an incoming feature to the corresponding feature of a contained component. In other words, these connections traverse up and down the containment hierarchy.

Other connections connect outgoing features of a component to incoming features of another component at the same level of the containment hierarchy, i.e., it connects sibling components. These connections occur at the highest level required for the connection declaration or at the top of the containment hierarchy required for the declaration.

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 **in** or **in out** event port in the enclosing component, or an **out** or **in out** event port of one of the subcomponents (see Section 11).

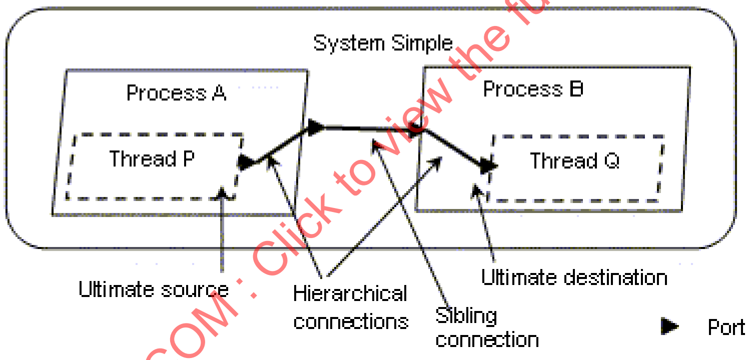


Figure 11 Semantic Port Connection

This section defines the concepts of departure and arrival times of port connection transmission for each of the *port connection categories*, i.e., for **data port** connections, **event port** connections, **event data port** connections, and **port group** connections. The transfer semantics between connected ports are defined such that the departure and arrival times of connection transmissions occurs in terms of deadline, execution completion, and dispatch times. These semantics ensure deterministic communication between periodic threads through data ports.

Syntax

```
port_connection ::=
    data_connection
  | event_connection
  | event_data_connection
  | port_group_connection
```

```
data_connection ::=
    [ defining_data_connection_identifier :]
    data port source_unique_port_identifier
        ( immediate_connection_symbol | delayed_connection_symbol )
        destination_unique_port_identifier
            [ { { property_association }+ } ]
            [ in_modes_and_transitions ] ;

immediate_connection_symbol ::= ->

delayed_connection_symbol ::= ->>

event_connection ::=
    [ defining_event_connection_identifier :]
    event port source_unique_port_identifier
        -> destination_unique_port_identifier
            [ { { property_association }+ } ]
            [ in_modes_and_transitions ] ;

event_data_connection ::=
    [ defining_event_data_connection_identifier :]
    event data port source_unique_port_identifier
        -> destination_unique_port_identifier
            [ { { property_association }+ } ]
            [ in_modes_and_transitions ] ;

-- connection between port groups of two subcomponents or between
-- a port group of a subcomponent and a port group in the component type
port_group_connection ::=
    [ defining_port_group_connection_identifier :]
    port group source_unique_port_group_identifier
        -> destination_unique_port_group_identifier
            [ { { property_association }+ } ]
            [ in_modes_and_transitions ] ;
```

```
port_connection_refinement ::=
    connection_identifier : refined to
        ( data port | event port | event data port | port group )
        ( ( { { property_association }+ }
          [ in_modes_and_transitions ] )
         | in_modes_and_transitions
        ) ;

unique_port_identifier ::=
        -- port in the component type
      component_type_port_identifier
    |
        -- port in a subcomponent
      subcomponent_identifier . port_identifier
    |
        -- port element in a port group of the component type
      component_type_port_group_identifier . element_port_identifier

unique_port_group_identifier ::=
        -- port group in the component type
      component_type_port_group_identifier
    |
        -- port group in a subcomponent
      subcomponent_identifier . port_group_identifier
    |
        -- port group element in a port group of the component type
      component_type_port_group_identifier . element_port_group_identifier
```

Naming Rules

The defining identifier of a defined port connection declaration must be unique in the local namespace of the component implementation with the connection subclause. For mode-specific connection declarations, as indicated by the `in_modes_and_transitions` subclause, a connection name may appear more than once.

The connection identifier in a port connection refinement declaration must refer to a named connection declared in an ancestor component implementation.

A source or destination reference in a port connection declaration must reference a port or port group declared in the component type, a port or port group of one of the subcomponents, or a port or port group that is an element of a port group in the component type.

Legality Rules

The ultimate source of a semantic port connection must be a feature of a thread, processor, or device. The source feature referenced in a port connection declaration must be a feature of a thread, thread group, process, processor, device, or system component. The ultimate destination of a semantic port connection must be a port of a thread, a processor, a device. If the ultimate destination is the result of a mode transition, the mode change is indicated by the mode subclause of the respective thread, thread group, process, system, device, bus, memory, or processor naming an **in event port** in one of its mode transitions. The destination feature referenced in a port connection declaration must be a feature of a thread, thread group, process, processor, device, or system component.

One end of the connection must be a thread. The other end may be a processor, a device, or a thead.

If the ultimate destination of a semantic port connection is the result of a mode transition, then the ultimate source must be an out event port.

If a semantic port connection is declared to apply to a particular mode, then the ultimate source and ultimate destination components must be part of that mode.

If a semantic port connection is declared to apply to 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.

The category of the port connection declaration must match the source and destination features as described in the following paragraphs. This implies that all connection declarations of a semantic connection must be of the same category.

The direction declared for the destination feature of a port connection declaration must be compatible with the direction declared for the source feature(s) as defined by the following rules:

- If the port connection declaration represents a connection between sibling components, then the source must be an **out** or an **in out** port and the destination must be an **in** or an **in out** port, or in the case of port group connections the source and destination port groups must be complements of each other (see Section 8.2).
- If the port connection declaration represents a connection between elements of two port groups in the component type, then source must be an **in** or an **in out** port and the destination must be an **out** or an **in out** port, or in the case of port group connections the source and destination port groups must be complements of each other (see Section 8.2).
- If the port connection declaration represents a connection up or down the containment hierarchy, then the source and destination must both be an **out** or an **in out** port, or both an **in** or **in out** port, or in the case of port group connections the port groups of the same port group type.

A data port cannot be the destination feature reference of more than one port connection declaration unless each port connection declaration is (are) contained in a different mode. In this case, the restriction applies for each mode.

The ultimate source and ultimate destination of a delayed port connection must be periodic threads.

For data and event data port connections, the data classifier of the source port must be identical to the data type of the destination port.

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.

For port group connections the following must hold:

- If the connection declaration represents a component connection between sibling components, the port group types must be complements as indicated with the **inverse of** statement in one of the two port group types.
- If the connection declaration represents a connection up or down the containment hierarchy, the port group types must be identical.

Standard Properties

```
Connection_Protocol: Supported_Connection_Protocols
Allowed_Connection_Binding_Class:
   inherit list of classifier(processor, bus, device)
Allowed_Connection_Binding: inherit list of reference (bus, processor, device)
Not_Collocated: list of reference (data, thread, process, system, connections)
Actual_Connection_Binding: inherit reference (bus, processor, device)
```

Semantics

A semantic port connection represents directed flow of data and control between two threads, between a processor and a thread, or a device and a thread. In the case of event port connections the ultimate destination can be a in a new mode.

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.

Data connections are restricted to 1-n connectivity, i.e., a data port can have multiple outgoing connections, but only one incoming connection. If the component with the destination data port has modes then this restriction applies to each mode. Port groups may have multiple outgoing and incoming connections unless any ports that are elements of a port group place additional restrictions.

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. Bi-directional flow between two components is represented by two connections between the **in out** ports of two components.

A port connection can be refined by adding property associations for the connection in a connection refinement declaration.

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 11.1.

While in a given mode, transmission over a port connection only occurs if the connection is part of the current mode.

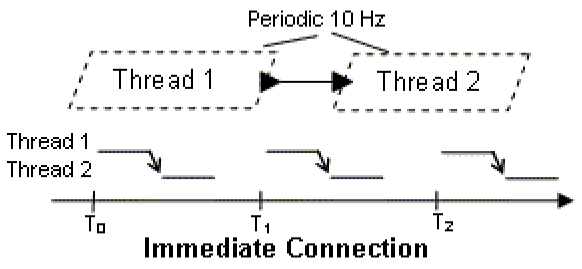
During a mode switch, transmission over a data port connection only occurs at the actual time of mode switch if the port connection is declared to apply to the transition between two specific modes. The actual mode switch initiates transmission. This allows data state to be transferred between threads active in different modes.

A data port connection is declared to be *immediate* ("**->**") or to be *delayed* ("**->>**"). A semantic data port connection is considered to be delayed if at least one of the connection declarations is declared to be delayed. Otherwise, the semantic data port connection is considered to be immediate. Typically, a delayed data connection is specified through the sibling connection declaration, i.e., the declaration at the top of the containment hierarchy of a semantic connection.

For immediate data port connections the data transmission is initiated when the source thread completes and enters the suspended state. Immediate data transfer only occurs when the periods of the sending and receiving thread align, i.e., their dispatch occurs logically simultaneous. The actual execution of the receiving thread is delayed until the sending thread completes execution. The content of the receiving thread's port variables is determined at the time of dispatch except for data ports that are connected by immediate connection. Their value is the data port value of the sending thread at the time of the sending thread's execution completion.

Immediate and delayed connections are illustrated in Figure 12. Thread 1 and Thread 2 are two periodic threads executing at a rate of 10Hz, i.e., they are logically dispatched every 100 ms. For immediate connection, shown on the left of the figure, the actual start of execution of the receiving thread (Thread 2) will be delayed after its dispatch event until the sending thread (Thread 1) completes execution and its **out** port data value has been transferred into the **in** port of the receiving thread. 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.

For delayed data port connections, the data transmission is initiated at 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. This is shown on the right of Figure 12. 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.



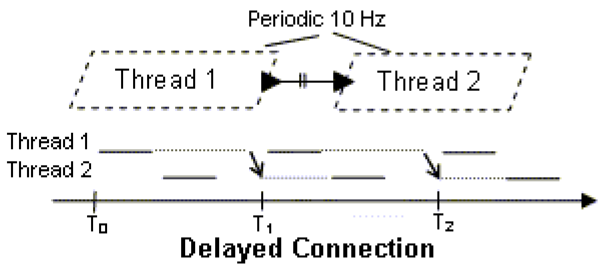


Figure 12 Timing of Immediate & Delayed Data Connections

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.

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 at all.

The semantics of immediate and delayed data transmission between periodic threads assures deterministic communication of state data. The alignment of transmission start and end times between the sending and receiving thread are statically known and are not affected by preemption of thread execution and variation in actual execution time.

NOTES:

Such deterministic communication cannot always be guaranteed if the transmission is initiated and completed by explicit send and receive service calls in the source text of the sending and receiving thread. If these calls are executed at the normal thread priorities, the time of actual data transfer through the send and receive call may vary and result in non-deterministic change in the send and receive order of two communicating threads.

If a processor or device is the data connection source, then the transmission is initiated and completed when the destination thread is dispatched.

For event and event data connections the transmission of control and data occurs immediately when the source thread executes a `Raise_Event` call.

If the event connection source is a device or processor, then the occurrence of an interrupt represents the initiation of an event transmission.

Transmission completion for event and event data connections results in queuing of the event or event data. It also represents the arrival of a dispatch request for an aperiodic or sporadic thread, if the thread's `Dispatch` port is not connected. For details on the content of port variables at the time of dispatch for periodic, aperiodic and sporadic threads see Section 8.1.

Within a synchronized system, an event arrives logically simultaneously at all ultimate connection destinations (see also Section 12.3).

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 11). Events that trigger mode transitions are not queued at event ports.

Processing Requirements and Permissions

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 physical system, a method of implementing systems must suppy a thead 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.

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.

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.

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.

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.

Examples

```
-- A simple example showing a system with two processes and threads.
-- The threads have a semantic connection.
-- The connection declarations follow the containment hierarchy.
data Alpha_Type
properties
   Source_Data_Size => 256 B;
end Alpha_Type;

port group xfer_plug
features
   Alpha : out data port Alpha_Type;
   Beta : in data port Alpha_Type;
end xfer_plug;

port 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 : port group xfer_plug;
end A;

process implementation A.Impl
subcomponents
   Producer : thread P.Impl;
   Result_Consumer : thread Q.Impl;
connections
   data port Producer.Data_Source -> Produce.Alpha;
   data 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 : port group xfer_socket;
end B;

process implementation B.Impl
```

```
subcomponents
   Consumer : thread Q.Impl;
   Result_Producer : thread P.Impl;
connections
   data port Consume.Alpha -> Consumer.Data_Sink;
   data 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
   port group pr_A.Produce -> pr_B.Consume;
end Simple.Impl;
```

9.1.2 Parameter Connections

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 **in** data or event data port or **in out** data or event data port of the containing thread to a subprogram call **in** or **in out** parameter. Parameter connections also specify connections from an **in** parameter or **in out** parameter of the containing subprogram to a subprogram call **in** or **in out** parameter, from a subprogram call **out** or **in out** parameter to a **out** or **in out** parameter of the containing subprogram, and from a subprogram call **out** or **in out** parameter to a subprogram call **in** or **in out** parameter or an **out** or **in out** data or event data port of the containing thread. In other words, the parameter connection declarations follow the containment hierarchy of subprogram calls nested in other subprograms. This is illustrated in Figure 13.

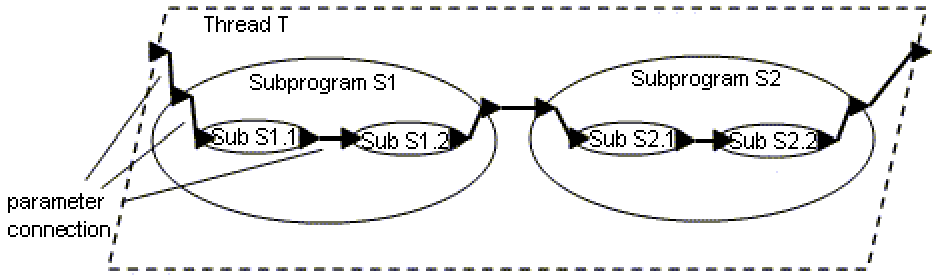


Figure 13 Parameter Connections

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

Syntax

```
parameter_connection ::=
    [ defining_parameter_connection_identifier :]
    parameter source_unique_parameter_identifier
        -> destination_unique_parameter_identifier
           [ { { property_association }+ } ]
           [ in_modes ] ;

parameter_connection_refinement ::=
    connection_identifier : refined to parameter
        { { property_association }+ }
        [ in_modes ] ;

unique_parameter_identifier ::=
        -- parameter in the thread or subprogram type
      component_type_parameter_identifier
    |
        -- parameter in another subprogram call
      subprogram_call_identifier . parameter_identifier
    |
        -- data or even data port in the thread type of the component type
      component_type_port_identifier
    |
        -- port element in a port group of the component type
    -- The port element must be a data or event data port
component_type_port_group_identifier . element_port_identifier
```

Naming Rules

The defining identifier of a defined parameter connection declaration must be unique in the local namespace of the component implementation with the connection subclause. For mode-specific parameter connections, as indicated by the `in_modes` subclause, a connection name may appear more than once.

The connection identifier in a parameter connection refinement declaration must refer to a named connection declared in an ancestor component implementation.

A source or destination reference in a parameter connection declaration must reference a parameter of a preceding 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, or a data port or event data port that is an element of a port group in the component type of the enclosing thread.

Legality Rules

Parameter connections must adhere to the following rule regarding their source and destination:

the source must be an **in** data or event data port or **in out** data or event data port of the containing thread and the destination a subprogram call **in** or **in out** parameter,

the source must be an **in** parameter or **in out** parameter of the containing subprogram and the destination a subprogram call **in** or **in out** parameter,

the source must be a subprogram call **out** or **in out** parameter and the destination a **out** or **in out** parameter of the containing subprogram,

the source must be a subprogram call **out** or **in out** parameter and the destination a subprogram call **in** or **in out** parameter or an **out** or **in out** data or event data port of the containing thread.

If the parameter connection declaration represents a parameter connection between sibling components, then the source must be an **out** or an **in out** parameter and the destination must be an **in** or an **in out** parameter. Furthermore, the source must be a parameter of a preceding subprogram call in the call sequence, and the destination must be a parameter of a succeeding subprogram call in the call sequence.

If a parameter connection is declared to apply to a particular mode, then the source and destination must be part of that mode.

A parameter cannot be the destination feature reference of more than one parameter connection declaration unless the source feature reference(s) of each parameter connection declaration is (are) contained in a different mode. In this case, the restriction applies for each mode.

The data classifier of the source port or parameter must be identical to the data type of the destination port or parameter.

Semantics

Parameter connections represent sequential flow of data through subprogram parameters in a sequence of subprogram calls being executed by a thread. Those calls may be performed locally, i.e., within the virtual address space of the containing process, or remotely by a synchronous call to a server subprogram in another thread. In the latter case, the parameter values are passed via a local subprogram proxy.

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.

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.

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.

The optional `in_modes` subclause specifies what modes the parameter connection is part of. The detailed semantics of this subclause are defined in Section 11.1.

9.1.3 Access Connections

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. These connections are *semantic access connections*. A semantic access connection is defined by a sequence of one or more individual access connection declarations that follow the component containment hierarchy in a fully instantiated system from an *ultimate source* to an *ultimate destination*.

The ultimate source of a semantic access connection is the data component or bus component that is being shared. The ultimate destination of an access connection is the component requiring the access without a contained subcomponent also requiring access. For data access connections this can be a thread or a subprogram call. For bus access connections the ultimate destination may be a processor, memory, or a device. The direction of the connection follows from the provider of access to the requirer of access. Figure 14 illustrates a semantic data connection from the data component D to thread Q.

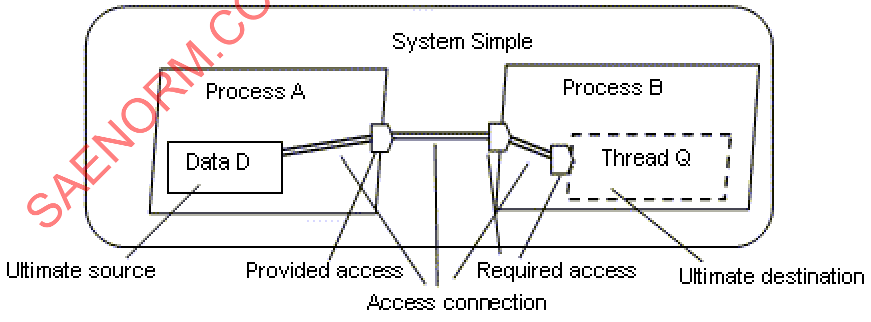


Figure 14 Semantic Access Connection

The flow of data of a semantic access connection is determined by the fact whether an ultimate destination has read access or write access to the shared component. The actual data flow is specified using the properties `Required_Access` or `Provided_Access`.

Syntax

```
access_connection ::=
     [ access_connection_identifier :]
     ( bus | data ) access unique_access_provider_identifier
         -> unique_access_requirer_identifier
            [ { { property_association }+ } ]
            [ in_modes ] ;

access_connection_refinement ::=
     connection_identifier : refined to ( bus | data ) access
         { { property_association }+ }
          [ in_modes ] ;

unique_access_provider_identifier ::=
         -- required access feature in the component type
       component_type_access_identifier
     |
         -- provided access in a subcomponent
       data_or_bus_subcomponent_identifier . access_identifier
     |
         -- data or bus subcomponent being accessed
       data_or_bus_subcomponent_identifier

unique_access_requirer_identifier ::=
         -- provided access feature in the component type
       component_type_access_identifier
     |
         -- required access in a subcomponent
       data_or_bus_subcomponent_identifier . access_identifier
```

Naming Rules

The defining identifier of a access connection declaration must be unique in the local namespace of the component implementation with the connection subclause. For mode-specific access connections, as indicated by the `in_modes` subclause, a connection name may appear more than once.

The connection identifier in an access connection refinement declaration must refer to a named connection declared in an ancestor component implementation.

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, or a data or bus subcomponent. A requirer reference in an access connection declaration must reference a requires access feature of a subcomponent or a provides access feature in the component type of the containing component.

Legality Rules

All access declarations forming a semantic data access connection must be data access declarations. All access declarations forming a semantic bus access connection must be bus access declarations.

The ultimate source of a semantic access connection must be data or bus subcomponent. The ultimate destination of a semantic data access connection must be a requires data access feature of a thread or a subprogram call without a containing subprogram call requiring the same data access. The ultimate destination of a semantic bus access connection must be a requires bus access feature of a processor, memory, or device subcomponent.

If a semantic access connection is declared to apply to a particular mode, then the ultimate source and ultimate destination must be part of that mode.

For access connections between access features, the direction declared for the destination feature must be compatible with the direction declared for the source feature(s) as defined by the following rules:

- If the access connection declaration represents an access connection between access features of sibling components, then the source must be a provides access or a data or bus component and the destination must be a requires access.

- If the access connection declaration represents a feature mapping up or down the containment hierarchy, then the source and destination must both be a requires access, both be a provides access, or the source a data or bus subcomponent and the destination a provides access.

A requires access cannot be the 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.

For access connections the data type of the provider access must be identical to the data type of the requires access.

If more than one access feature in a semantic access connection has an access `Required_Access` or `Provided_Access` 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.

Semantics

An access connection represents access to a shared data component by concurrently executing threads or by subprograms executing within thread. A bus access connection represents communication between processors, memory, and devices by accessing a shared bus.

Access connections are restricted to 1-n connectivity, i.e., a data or bus component can have multiple outgoing access connections, but a **requires** access feature can only have one incoming connection.

The actual data flow is determined by the value of the `Required_Access` or `Provided_Access` property. 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 optional `in_modes` subclause specifies what modes the access connection is part of. The detailed semantics of this subclause are defined in Section 11.1.

9.2 Flows

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.

End-to-end flows are represented by *flow specification*, *flow implementation*, and *end-to-end flow* declarations.

A flow specification declaration in a component type specifies an externally visible flow through a component's ports, port groups, or parameters. 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*.

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 in port to the flow specification out port. This is illustrated in Figure 15. 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.

The system implementation for system S1 is shown on the right of Figure 15. 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.

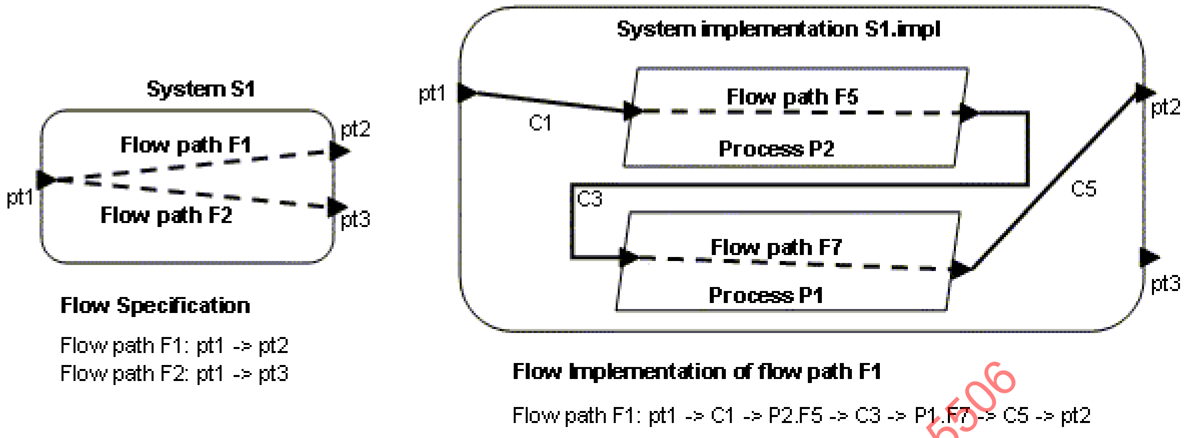


Figure 15 Flow Specification & Flow Implementation

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 flow implementation in the system hierarchy that is the root of all threads, 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 15, 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. The identified subcomponent of the final referenced subcomponent flow specification contains the last system component of the end-to-end flow.

9.2.1 Flow Specifications

A flow specification declaration indicates that information logically flows from one of its incoming ports, parameters, or port groups to one of its outgoing ports, parameters, or port groups. 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*. In the case of port groups, there is a flow from a port group to its inverse.

Multiple flow specifications can be defined involving the same ports. For example, data coming in through an **in** port group is processed and derived data from one of the port 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 flow_feature_identifier
   [ { { property_association }+ } ] ;

flow_sink_spec ::=
    defining_flow_identifier : flow sink flow_feature_identifier
   [ { { property_association }+ } ] ;

flow_path_spec ::=
    defining_flow_identifier : flow path source_flow_feature_identifier ->
                                sink_flow_feature_identifier
   [ { { property_association }+ } ] ;

flow_source_spec_refinement ::=
    defining_flow_identifier :
        refined to flow source { { property_association }+ } ;

flow_sink_spec_refinement ::=
    defining_flow_identifier :
        refined to flow sink { { property_association }+ } ;

flow_path_spec_refinement ::=
    defining_flow_identifier :
        refined to flow path { { property_association }+ } ;

flow_feature_identifier ::=
    port_identifier
    | parameter_identifier
    | port_group_identifier
    | port_group_identifier . port_identifier
```

Naming Rules

The defining flow identifier of a flow specification must be unique within the interface name space of the component type.

The flow feature identifier in a flow path must refer to a port, parameter, or port group in the component type, or to a port or port group contained in a port group in the component type.

The defining flow identifier of a flow specification refinement must refer to a flow specification or refinement in an ancestor component type.

Legality Rules

The direction declared for the destination of a flow path specification declaration must be compatible with the direction declared for the source as defined by the following rules:

- If the source is a port or parameter, its direction must be must be an **in** or an **in out**.
- If the destination is a port or parameter, its direction must be an **out** or an **in out**.

The direction declared for the destination port or parameter of a flow source specification declaration must be **out** or **in out**.

The direction declared for the source port or parameter of a flow source specification declaration must be **in** or **in out**.

Standard Properties

```
Latency: Time
Throughput: Data_Volume
```

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 10).

Semantics

A flow specification declaration represents a logical flow originating from within a component, flowing through a component, or ending within a component.

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 and between ports, parameters and port 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

```
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 event 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;
```

9.2.2 Flow Implementations

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 source port and ends with the destination 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.

By declaring flow specifications explicitly we clearly specify the expectations of a component, for both the user of a component and the implementer of a component. Compliance with the specifications can be checked separately from both perspectives.

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 -> }*
      flow_feature_identifier

flow_sink_implementation ::=
    flow_identifier : flow sink
      flow_feature_identifier
      { -> connection_identifier -> subcomponent_flow_identifier }*

flow_path_implementation ::=
    flow_identifier : flow path
      source_flow_feature_identifier
      [ { -> connection_identifier -> subcomponent_flow_identifier }+
         -> connection_identifier ]
      -> sink_flow_feature_identifier

flow_implementation_refinement ::=
    flow_source_implementation_refinement
    | flow_sink_implementation_refinement
    | flow_path_implementation_refinement

flow_source_implementation_refinement ::=
    flow_identifier :
        refined to flow source
        ( { { property_association }+ } [ in_modes_and_transitions ]
        | in_modes_and_transitions
        ) ;

flow_sink_implementation_refinement ::=
    flow_identifier :
        refined to flow sink
        ( { { property_association }+ } [ in_modes_and_transitions ]
        | in_modes_and_transitions
        ) ;

flow_path_implementation_refinement ::=
    flow_identifier :
        refined to flow path
        ( { { property_association }+ } [ in_modes_and_transitions ]
        | in_modes_and_transitions
        ) ;


subcomponent_flow_identifier ::=
    subcomponent_identifier . flow_spec_identifier
```

Naming Rules

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.

The flow feature identifier in a flow implementation must refer to a port, parameter, or port group in the component type, or to a port or port group contained in a port group in the component type.

The subcomponent flow identifier of a flow implementation must name a flow specification in the component type of the named subcomponent.

The connection identifier in a flow implementation must refer to a connection in the component implementation.

The defining flow identifier of a flow implementation refinement must refer to a flow implementation or refinement in an ancestor component implementation.

Legality Rules

The source of a connection named in a flow implementation declaration must be the same as the source flow feature of the flow implementation or as the destination of the directly preceding subcomponent flow specification.

The destination of a connection named in a flow implementation declaration must be the same as the destination flow feature of the flow implementation or as the source of the directly succeeding subcomponent flow specification.

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.

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.

In a complete specification, if a system, process, or thread group component implementation contains a flow implementation declaration, then the flow implementation must include flow specifications through named thread, processor, or device subcomponents.

If the category of the component type containing a flow specification declaration is thread or subprogram, and a component implementation of the component type does not contain subprogram calls, then the flow specification represents its implementation and an explicit flow implementation declaration is not required.

Standard Properties

```
Latency: Time
Throughput: Data_Volume
```

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

A flow implementation declaration represents the realization of a flow specification in the given component implementation. A flow implementation may be declared to be mode-specific.

A flow path implementation starts with the port named in the corresponding flow specification, passes through zero or more subcomponents, and ends with the port named in the corresponding flow specification (see Figure 15). A flow source implementation ends with the port named in the corresponding flow specification. A flow sink implementation starts with the port named in the corresponding flow specification. A flow path implementation may specify a flow that goes directly from a flow source to a flow destination without any connections in between.

A flow implementation within a thread may be modeled as flow through subprogram calls via their parameters.

A flow through a 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 and between ports and port 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.

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 11.1.

Examples

```
-- 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
   C: thread baz.basic;
   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: data port signal -> A.p1;
   conn2: data port A.p2 -> B.p1;
   conn3: data port B.p2 -> result1;
   conn4: data port A.p2 -> C.p1;
   conn5: data port C.p2 -> result2;
   conn6: data port A.p3 -> status;
   connToThread: event port initcmd -> C.reset;
flows
   Flow1: flow path
            signal -> conn1 -> A.fs1 -> conn2 ->
            B.fs1 -> conn3 -> result2;
   Flow2: flow path
            signal -> conn1 -> A.fs1 -> conn4 ->
            C.fs1 -> conn5 -> 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 -> connect6 -> status;
end foo.basic;
```

9.2.3 End-To-End Flows

An end-to-end flow represents a logical flow of information from a source to a destination through a sequence of threads that process and possibly transform the information. In a complete specification, the source and destination can be threads, devices, and processors.

Syntax

```
end_to_end_flow_spec ::=
    defining_end_to_end_flow_identifier : end to end flow
      start_subcomponent_flow_identifier
      { -> connection_identifier
        -> flow_path_subcomponent_flow_identifier }*
      -> connection_identifier -> end_subcomponent_flow_identifier
    [ { ( property_association }+ } ]
    [ in_modes_and_transitions ] ;


end_to_end_flow_refinement ::=
    defining_end_to_end_identifier :
        refined to end to end flow
        ( { { property_association }+ } [ in_modes_and_transitions ]
        | in_modes_and_transitions
        ) ;
```

Naming Rules

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. For mode specific end-to-end flows, as indicated by the `in_modes_and_transitions` subclause, an end-to-end flow identifier may appear more than once.

The connection identifier in an end-to-end flow declaration must refer to a connection in the component implementation.

The subcomponent flow identifier of an end-to-end flow declaration must name a flow specification in the component type of the named subcomponent.

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

The flow specifications identified by the *flow_path*_`subcomponent_flow_identifier` must be flow paths.

The *start*_`subcomponent_flow_identifier` must refer to a flow path or a flow source.

The *end*_`subcomponent_flow_identifier` must refer to a flow path or a flow sink.

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

```
Expected_Latency: Time
Actual_Latency: Time
Expected_Throughput: Data_Volume
Actual_Throughput: Data_Volume
```

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

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 end-to-end flow starts with a subcomponent flow specification, followed by zero or more connections and subcomponent flow specificaitons, and ends with a connection and a subcomponent flow specification. The actual end-to-end flow starts from a device, processor, or thread, follows semantic connections to intermediate threads and ends with a thread, device or processor. If the start or end point of an end-to-end flow is a thread, its contribution to the flow may be limited to a partial execution by specifying a flow implementation through a subset of its subprogram calls.

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 11.1.

Examples

```
-- 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
   C: thread baz.basic;
   B: thread baz.basic;
   -- baz has a flow path fs1
   -- baz has a flow sink fsink
connections
   conn1: data port signal -> A.p1;
   conn3: data port C.p2 -> result1;
   conn4: data port A.p2 -> C.p1;
   conn5: event port A.p3 -> Status;
   connToThread: event port initcmd -> C.reset;
flows
   Flow1: flow path
            signal -> conn1 -> A.fs1 -> conn4 ->
            C.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 -> connect5 -> status;
   -- an end-to-end flow from a source to a sink
   ETE1: end to end flow
            A.fs2 -> conn4 -> C.fsink;
   -- an end-to-end flow where the end points are not sources or sinks
   ETE2: end to end flow
            A.fs1 -> conn4 -> C.fs1;
end foo.basic;
```

10 Properties

A property provides information about component types, component implementations, subcomponents, features, connections, flows, modes, and subprogram calls. A property has a name, a type, and a value. The property name 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.

A property set contains declarations of property types and property names 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 name declaration within a property set indicates the component types, component implementations, subcomponents, features, connections, flows, modes, and subprogram calls, for which this property applies.

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 name may have a default expression.

10.1 Property Sets

A property set defines a named group of property types, property names, and property constant values.

Syntax

```
property_set ::=
    property set defining_property_set_identifier is
       { property_type_declaration
         | property_name_declaration
         | property_constant }+
    end defining_property_set_identifier ;
```

Naming Rules

Property set defining identifiers must be unique in the global namespace.

The defining identifier following the reserved word **end** must be identical to the defining identifier following the reserved word **property set**.

Associated with every property set is a *property set namespace* that contains the defining identifiers for all property types declared within that property set. This means that properties with the same identifier can be declared in different property sets.

A property or property type declared in a property set is 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.

10.1.1 Property Types

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_designator ;


property_type_designator ::=
      property_type | unique_property_type_identifier


property_type ::=
        aadlboolean | aadlstring
      | enumeration_type | units_type
      | number_type | range_type
      | classifier_type
      | reference_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 [ ( component_category { , component_category }* ) ]


reference_type ::=
      reference  [ ( referable_element_category
                     { , referable_element_category }* ) ]


referable_element_category ::=
        component_category | connections | server subprogram


unique_property_type_identifier ::=
     [ property_set_identifier :: ] property_type_identifier
```

Naming Rules

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.

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. Otherwise, the property type identifier must appear in the property set namespace.

An enumeration type introduces an enumeration namespace. The enumeration literal identifiers in the enumeration list declare a set of enumeration literals. They must be unique within this namespace.

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.

The units identifier to the right of a **=>** must refer to a unit identifier defined in the same units type declaration.

Legality Rules

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.

Range values should always be declared with unit literals if the property requires a unit literal.

The unique property constant identifier in an integer range must represent an integer constant.

A boundless range type may be declared such that the actual range declarations have no limit on the upper and lower bound.

The unique property constant identifier in a real range must represent a real constant.

If the property requires a unit , then the unit must be specified for both lower and upper bound.

Semantics

A property type declaration associates an identifier with a property type.

The **aadlboolean** property type represents the two values, true and false.

The **aadlstring** property type represents all legal strings of the AADL.

An **enumeration** property type represents an explicitly listed set of enumeration identifiers as the set of legal values.

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 the previous unit to determine the value in terms of the specified measurement unit.

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.

An **aadlinteger** property type represents an integer value or an integer value and its measurement unit. If an 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 an 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.

The **range** 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.

A **classifier** property type represents the subset of syntactically legal component classifier references whose category matches one of component categories in the specified list. If the category list is absent, all component classifier references are acceptable.

A **reference** type (indicated by the reserved word **reference**) represents the subset of syntactically legal references to those components whose category matches one of component categories in the specified list, or to connections, or to server subprogram features. If the category list is absent, all components, connections, and server subprograms are acceptable.

NOTES:

The classifier and reference property types support the specification of properties representing binding constraints.

Examples

```
Length_Unit : type units ( mm, cm => mm * 10,
                           m => cm * 100, km => m * 1000 );
OnOff : type aadlboolean;
Car_Length : type aadlreal 1.5 .. 4.5 units ( meter );
Speed_Range : type range of aadlreal 0 .. 250 units ( kph );
```

10.1.2 Property Names

All property names that appear in a property association list must be declared with property name declarations inside a property set. Property names are typed and are defined for specific component, port, port group, subprogram, access, mode, flow, and connection categories.

Syntax

```
property_name_declaration ::=
    defining_property_name_identifier : [ access ] [ inherit ]
        ( single_valued_property | multi_valued_property )
        applies to (
               ( property_owner_category { , property_owner_category }*
               | all )
        ) ;

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_category ::=
      component_category [ classifier_reference ]
    | mode | port group | flow
    | [ event ] [ data ] port
    | server subprogram | parameter
    | [ connection_type ] connections

connection_type ::=
    port group | [ event ] [ data ] port | access | parameter
```

Naming Rules

All property name defining identifiers 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 names declared within that property set.

Legality Rules

The reserved word **access** is only permitted for property name declarations whose **applies to** property category list contains categories of subcomponents that can be required or provided subcomponents. These categories are data and bus.

Semantics

A property name declaration introduces a new property by a name 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 component categories, specific component classifiers, or for port, port group, flow, subprogram, mode, and connection categories that are listed after the **applies to** in the `Property_Owner_Category` list. This indicates that component classifiers and subcomponents corresponding to the specified category, ports, port groups, flows, subprograms, modes, and connections can have property associations for such a property. If the category specification includes a classifier, and the classifier is a component type, then the property applies to both the type and implementation of the specified component category. In the case of a component implementation classifier, the property applies only to the implementation. The reserved word **all** in the **applies to** statement indicates that the property applies to any `Property_Owner_Category`.

A property declared with the reserved word **access** is associated with the access to a subcomponent rather than the data component itself. For example, two components can require access to a data component, one requiring read-only access, while the other requires write-only access.

A property declared 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 10.3.

A property name declared without a default value is considered undefined (see also Section 10.3). A property name declared to have a list of values is considered to have an empty list if no default value is declared.

Examples

```
Wheel_speed : aadlinteger 0 rpm .. 5000 rpm units ( rpm )
                                            applies to ( system );
```

10.1.3 Property Constants

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
            ( ( aadlinteger | aadlreal
                [ units_unique_property_type_identifier ] )
                | aadlstring | aadlboolean
                | enumeration_unique_property_type_identifier
                | integer_range_unique_property_type_identifier
                | real_range_unique_property_type_identifier
                | integer_unique_property_type_identifer
                | real_unique_property_type_identifer )
        => constant_property_value ;

multi_valued_property_constant ::=
    defining_property_constant_identifier : constant list of
            ( ( aadlinteger | aadlreal
                [ units_unique_property_type_identifier ] )
                | aadlstring | aadlboolean
                | enumeration_unique_property_type_identifier
                | integer_range_unique_property_type_identifier
                | real_range_unique_property_type_identifier
                | integer_unique_property_type_identifer
                | real_unique_property_type_identifer )
        => ( [ constant_property_value { , constant_property_value }* ] ) ;
```

```
constant_property_value ::=
    string_literal
    | signed_integer
    | signed_real
    | boolean_value
    | enumeration_identifier
    | signed_aadlinteger .. signed_aadlinteger [ delta signed_aadlinteger ]
    | signed_aadlreal .. signed_aadlreal [ delta signed_aadlreal ]

unique_property_constant_identifier ::=
    value ( [ property_set_identifier :: ] property_constant_identifier )
```

Naming Rules

The defining property constant identifier must be distinct from all other property constant identifiers, property name identifiers, and property type identifiers in the namespace of the property set that contains the property constant declaration.

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

If a property constant declaration has more than one property expression, it must contain the reserved words **list of**.

The property type of the property constant declaration must match the property type of the constant property value.

If the constant property value is an integer or real value with a unit identifier, then the property type specification of the property constant must include a units identifier.

Semantics

Property constants allow integer, real, and string values to be known by symbolic name and referenced by that name in property expressions. This reference is expressed by the construct **value**() resulting in the value of the constant to be used instead of the reference.

Examples

```
Max_Threads : constant aadlinteger => 256;
```

10.2 Predeclared Property Sets

There is a standard predeclared property set named `AADL_Properties`, which is part of every AADL specification. In addition, there is a set of enumeration property types and property constants for which enumeration literals and values can be defined for different AADL specifications. This set of property types is declared in a property set named `AADL_Project`. 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. The `AADL_Properties` and `AADL_Project` property sets are implicitly a part of every AADL specification.

The property types, property names, and property constants of these predeclared property sets can be named without property set name qualification.

```
property set AADL_Properties is
   -- See Appendix A.1
end AADL_Properties;


property set AADL_Project is
   -- See Appendix A.2
end AADL_ Project;
```

Naming Rules

The predeclared property sets `AADL_Properties` and `AADL_Project` share a property set namespace.

Legality Rules

The `AADL_Properties` property set cannot be modified.

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

Additional property name declarations may not be inserted into the standard predeclared property set `AADL_Properties`. Different property set declarations must be used for nonstandard property names.

Providers of AADL processing methods may modify the standard property type declarations in `AADL_Properties` to allow additional values for a specific property name. For example, additional enumeration identifiers beyond those listed in this standard may be added.

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.

Additional property sets that may be suitable for a wide variety applications may be defined in an Annex. 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.

10.3 Property Associations

A property association correlates a property value or list of property values with a property name resulting from evaluation of 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. Subcomponents can also declare contained property associations of subcomponents contained in them. Contained property associations permit separate property values to be associated with every component in the system instance hierarchy (see Section 12.1).

Syntax

```
property_association ::=
  [ property_set_identifier :: ] property_name_identifier ( => | +=> )
      [ constant ] property_value
      [ in_binding ]
      [ in_modes ] ;

access_property_association ::=
  [ property_set_identifier :: ] property_name_identifier ( => | +=> )
      [ constant ] access property_value
      [ in_binding ]
      [ in_modes ] ;

contained_property_association ::=
  [ property_set_identifier :: ]
  property_name_identifier ( => | +=> )
     [ constant ] property_value
     applies to contained_unit_identifier { . contained_unit_identifier }*
     [ in_binding ]
     [ 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
        | memory_classifier_reference
        | bus_classifer_reference
```

Naming Rules

A property name consists of an optional property set identifier followed by a property identifier, separated by a double colon ("::").

The property set identifier, if present, must appear in the global namespace and must be the defining identifier in a property set declaration.

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 the predeclared property sets `AADL_Properties` or `AADL_Project`.

A property name may appear in the property association clause of a component type, component implementation, subcomponent, feature, flow, connection, mode, or subprogram call only if the respective AADL model element is listed in the **applies to** list of the property name declaration.

The dot-separated identifier sequence following the reserved words **applies to** of a contained property association identifies a component, feature, flow, connection, or mode in the containment hierarchy, for which the property value holds. The root of this path is the subcomponent or the component implementation with the contained property association declaration. The path consists of a sequence of zero or more subcomponent identifiers followed by a subcomponent, feature, flow, connection, or mode identifier. A port in a port group is identified by the port group identifier and the port identifier.

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, memory, or bus. If a property association list contains both binding-specific associations and an association without an **in binding** statement, then the latter applies to all bindings not explicitly declared in **in binding** statements.

If a property association has an **in modes** statement, the property value is mode-specific. The property value applies if one of the specified modes is active. If a property association list contains both mode-specific associations and an association without an **in modes** statement, then the latter applies to all associations not explicitly declared in **in modes** statements.

A property association list must have at most one property association for the same property name. In case of mode-specific and binding-specific property associations, there must be at most one association for each mode and binding.

Legality Rules

The property named by a property association must list the category of the component type, component implementation, subcomponent, feature, connection, flow, or mode the property association is declared for in its `Property_Owner_Category` list (see Section 10.1.2).

If a property association is declared within a package, the property value applies to all component classifier declarations contained in the package for which the property is valid.

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.

If the property declaration for the associated property name 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 name 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.

The property association operator **+=>** must only be used if the property declaration for the associated property name contains the reserved words **list of**. Furthermore, the property association may not have an **in modes** or **in binding** statement.

The property association operator **+=>** may not be used in contained property associations.

In a property association, the type of the evaluated property expression must match the property type of the named property.

A property value declared by a property association with the reserved word **constant** cannot be changed.

The semantics rules below for determining the value of a property impose a precedence on the property associations for a property. A property association with the reserved word **constant** must be the highest priority association.

The reserved word **access** is only permitted and is required in property associations declared in required and provided access subcomponent declarations and refinements.

The unique component type identifiers in the **in binding** statement must refer to component types of the categories **processor**, **memory**, or **bus.**

Property associations declared as part of a component type declaration, port group type declaration, a feature or feature refinement declaration in a component type or port group type, or as part of a feature refinement in the refines type clause of a component implementation are not permitted to have an **in modes** statement as the scope of modes is limited to component implementations. A feature refinement in the refines type clause of a component implementation must not inherit a modal property association from its component implementation.

Semantics

Property associations determine the property value of the component instances and their feature, connection, flow, and mode instances in the system instance hierarchy (see Section 12.1). The property association of a component type, component implementation, subcomponent, feature, flow, connection, mode, or subprogram call determines the property value of all instances derived from the respective declaration.

If a property association is declared within a package, the property value applies to all component classifier declarations contained in the package for which the property is valid.

The value of a property is determined through evaluation of the property expression. Property associations are declared in the properties subclause of component types and component implementations. They are also declared as part of feature declarations in component types, as part of subcomponent, connection, flow and mode declarations in component implementations. Contained property associations declared with subcomponents can represent separate property values for different instances of subcomponents, their features, connections, flows and modes that are contained in the subcomponent. Contained property associations can also be used to record system instance specific property values for all components, features, connections, flows, and modes in a system instance. This permits AADL analysis tools to record system instance specific information about a physical system separate from the declarative AADL specification. 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 for the same system.

The property value is determined according to the following rules, which impose a precedence on the property associations for a particular property. The earlier a property association for the given property is encountered by the rules, the higher it's precedence.

If a property value is not present after applying all of the rules below, it is determined by the default value of its property name declaration. If not present in the property name declaration, the property value is undefined.

For component types and port group types, 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 component type or port group type with its property association. If not present and the component type or port group type is declared in the private section of a package, then the property value is determined by its association in the property subclause of the private section. If not present in the private section, it is determined by its association in the property subclause of the package's public section. If the component type or port group type is declared in the public section of a package, the property value is determined by its association in the public section of the package. Otherwise, it is considered not present.

For component implementations and port groups, 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 component implementation or port group with its property association. If not present, it is determined by the property value of the component implementation's component type according to the component type rules.

For subcomponents, the property value of a property is determined by its property association in the subcomponent declaration. If not present and the subcomponent is refined, then the property value is determined by a property association in the subcomponent declaration being refined; this is done recursively along the refinement sequence. If not present in the subcomponent, it is determined by the subcomponent's component classifier reference according to the respective component implementation or component type rules described above. If not present and the property name has been declared as **inherit**, it is determined by the property value of the component implementation that contains the subcomponent declaration according to the component implementation rules. Otherwise, it is considered not present.

For modes, connections, or flow sequences the property value of a property is determined by its property association in the mode, connection, or flow sequence declaration. If not present and the mode, connection, or flow sequence is refined, then the property value is determined by a property association in the mode, connection, or flow sequence declaration being refined; this is done recursively along the refinement sequence. If not present and the property name has been declared as **inherit**, it is determined by the property value of the component implementation that contains the mode, connection, or flow sequence declaration according to the component implementation rules. Otherwise, it is considered not present.

For subprogram calls in call sequences, the property value of a property is determined by its property association in the subprogram call. If not present and the called subprogram name is a subprogram classifier reference, the property value is determined by the subprogram classifier according to the component implementation or component type rules described above. If not present and the called subprogram name is a subprogram feature reference in a data component, the property value is determined by the subprogram feature according to the feature rules described below. If not present and the property name has been declared as **inherit**, it is determined by the property value of the component implementation that contains the subprogram call according to the compoment implementation rules. Otherwise, it is considered not present.

For features in a component type or port group type, or flow specifications in a component type, the property value of a property is determined by its property association in the feature or flow specification declaration. If not present and the feature or flow specification is refined, then the property value is determined by a property association in the feature or flow specification declaration being refined; this is done recursively along the refinement sequence. For subprogram, server subprogram, and port group features, if not present and the feature references a subprogram classifier or port group type reference, the property value is determined by the subprogram component classifier reference or port group type according to the respective component implementation, component type, or port group type rules described above. If not present and the feature references a subprogram feature in a data component type, the property value is determined by the subprogram feature according to the feature rules. If not present and the property name has been declared as **inherit**, then it is determined by the property value of the component type or port group type that contains the feature or flow specification declaration according to the respective component type or port group type rules. Otherwise, it is considered not present.

For features in a refines type clause of a component implementation, the property value of a property is determined by its property association in the feature refinement declaration If not present, then the property value is determined by a property association in the feature declaration being refined; this is done recursively along the refinement sequence. If not present and the property value has been declared **inherit**, it is determined by the property value of the component implementation according to the component implementation rules. Otherwise, it is considered not present.

For component, feature, connection, flow, or mode 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, or mode. If not present, then the property value is determined by the respective subcomponent, mode, connection, feature declaration that results in the instance according to the rules above. If not present and the property name has been declared as **inherit**, then it is determined by the property value of the first containing component in the containment hierarchy of the system instance. Otherwise, it is undefined.

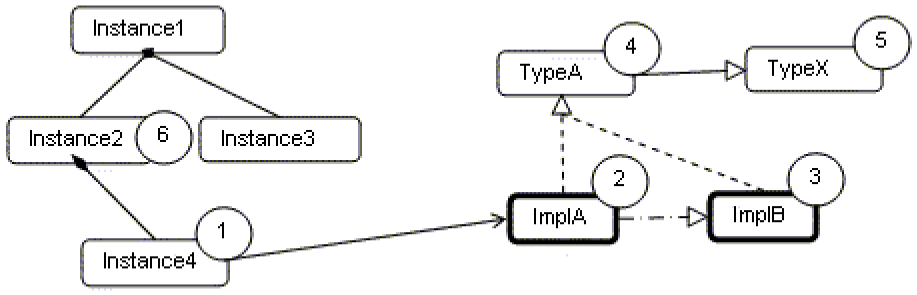


Figure 16 Property Value Determination

Figure 16 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.

Two property association operators are supported. The operator **=>** results in a new value for the property. The operator **+=>** results in the addition of a value to a property value list. More specifically, a property association via the operator **=>** replaces any associations of lower precedence according to the above rules. A property association via the operator **+=>** appends to the value determined by the association immediately preceding it according to the order imposed by the above rules.

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.

If a property association is declared with the reserved word **constant**, then the following rules apply:

For subcomponents, connections, flow sequences, and modes, any refinements cannot contain a property association for this property.

For features or flow specifications, any refinements cannot contain a property association for this property.

For component implementations, any component implementation extension, or any subcomponents referencing the component implementation or any of its descendents as component classifier cannot contain a property association for this property.

For port groups, any port group refinement cannot contain a property association for this property.

For component types, any component type extension, any component implementation, any subcomponent, or any subprogram or server subprogram feature referencing the component type or any of its descendents cannot contain a property association for this property.

For port group types, any port group type extension or any port group feature referencing the port group type or any its descendents cannot contain a property association for this property.

A property association declared with the reserved word **access** applies to the access to an actual subcomponent represented by the provided or required subcomponent access rather than the actual subcomponent itself. Two different components sharing access to a component may have different access property associations.

The optional `in_modes` subclause specifies what modes the property association is part of. The detailed semantics of this subclause are defined in Section 11.1.

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.

10.4 Property Expressions

A property expression represents the value that is associated with a property through a property association. The type of the value resulting from the evaluation of the property expression must match the property type declared for the property name.

Syntax

```
property_expression ::=
      boolean_term
    | real_term
    | integer_term
    | string_term
    | enumeration_term
    | real_range_term
    | integer_range_term
    | property_term
    | component_classifier_term
    | reference_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

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 ::=
    value ( [ property_set_identifier :: ] property_name_identifier )

property_constant_term ::=
    value ( [ property_set_identifier :: ] property_constant_identifier )

component_classifier_term ::=
    component_category
    [ unique_component_type_identifier
     [ . component_implementation_identifier  ] ]

reference_term ::=
    reference
        subcomponent_identifier { . subcomponent_identifier }*
        | { subcomponent_identifier. }+ connection_identifier
        | { subcomponent_identifier . }+ server_subprogram_identifier
```

NOTES:

Boolean operators have the following decreasing precedence order: (), **not**, **and**, **or**.

Naming Rules

The component type identifier or component implementation name of a subcomponent classifier reference must appear in the anonymous namespace or in the namespace of the specified package.

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.

Legality Rules

If the base type of a property number type or range type is integer, then the numeric literals must be integers.

The type of a property named in a property term must match the type of the property name in the property association.

The type of a property constant named in a property constant term must match the type of the property name in the property association.

Property name references in property expressions cannot be circular. If a property has a property expression that refers to a property name, then that property's expression evaluation cannot directly or indirectly depend on the value of the original property.

Semantics

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.

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 subexpressions. 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

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.

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.

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. Properties with range terms as values typically represent range and increment constraints on data streams communicated through ports.

String terms are of **aadlstring** property type. A string literal evaluates to the string of characters denoted by that literal.

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 **value**(`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.

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.

Component classifier terms are of the property type component **classifier**. They evaluate to a component category and an optional component classifier reference.

Reference terms are of **reference** property type and evaluate to a reference. This reference may be a reference to a component in the component containment hierarchy or to a connection or server subprogram contained in a component.

- For property associations of component implementations, the first identifier in the reference must appear as a subcomponent identifier in the local namespace of the component implementation to which the property association belongs.
- For property associations of subcomponents, the first identifier must appear as a subcomponent identifier within the local namespace of the component implementation that is referenced as the component classifier of the subcomponent.
- Subsequent identifiers must appear in the namespace of the component implementation associated with the subcomponent identified by the preceding identifier and must map to subcomponent declarations.

The entire reference evaluates to the component, connection, or server subprogram feature identified by the last identifier in that name. In other words, the sequence of identifiers specifies a path down the containment hierarchy starting with the component in the context of which the property association is declared.

NOTES:

Expressions of the property type **reference** or **classifier** are provided to support the description of binding constraints and of binding-specific property expressions.

Processing Requirements and Permissions

A method of processing specifications may define additional rules to determine if an expression value is legal for a property name, 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 name.

If an associated expression or default value is not specified for a property name, 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 may accept a specification in which that property has no associated value.

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.

Examples

```
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;

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_A;
end Total_System.SW_HW;
```

11 Operational Modes

Modes represent the operational states of software, execution platform, and compositional components in the modeled system. A component can have mode-specific property values. A component can also have mode-specific configurations of subcomponents and connections. Mode transitions model dynamic operational behavior that represents switching between configurations and changes in components internal characteristics.

This section defines modes and mode transitions to support the modeling of operational modes.

11.1 Mode

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 communicating threads 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 events 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 mode transition specifies possible runtime passage from one state or condition to another. Such transitions are triggered by events. 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 data, mode transitions model the changeover between modes that are encoded in the source text and may result in different associated property values.

Syntax

```
mode ::=
    defining_mode_identifier : [ initial ] mode
        [ { { mode_property_assocation }+ } ];


mode_transition ::=
    source_mode_identifier { , source_mode_identifier }*
        -[ unique_port_identifier { , unique_port_identifier }* ]->
        destination_mode_identifier ;


mode_refinement ::=
    defining_mode_identifier : refined to mode
        { { mode_property_assocation }+ } ;
```

```
in_modes ::=
    in modes ( ( mode_identifier { , mode_identifier }*
            | none ) )


in_modes_and_transitions ::=
    in modes ( ( mode_or_transition { , mode_or_transition }*
            | none ) )


mode_or_transition ::=
        mode_identifier | ( old_mode_identifier -> new_mode_identifier )
```

Naming Rules

The defining mode identifiers must be unique within the local namespace of the component implementation.

The identifiers in a mode transition that refer to modes must exist in the local namespace of the component implementation that contains the mode subclause. In other words, only modes declared in the component implementation that contains the mode transition or in any of its extension ancestors can be referenced.

The mode identifiers named in an **in modes** statement must refer to modes declared in the mode subclause of the component implementation that contains the subcomponent declaration, connection declaration, or property association with the **in modes**, or any of its extension ancestors. In other words, subcomponents, connections, and property associations can only be applicable to modes of the component implementation they are contained in.

The same mode or mode transition must not be named in the **in modes** statement of different mode-specific declarations of the same subcomponent, call sequence, flow implementation, and property association.

An **in modes** statement in a refinement declaration may be used to specify mode membership to replace the one, if any, in the declaration being refined.

The old and new mode identifier in a `mode_or_transition` clause must not be the same.

Legality Rules

A mode can be declared in data, thread, thread group, process, system, processor, bus, memory, and device component implementations.

If a component implementation contains mode declarations, one of those modes must be declared with the reserved word **initial**. If the component implementation extends another component implementation, the initial mode may have been declared in one of the ancestor component implementations.

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 associated with a single event arrival, which can cause transition to another mode.

The unique port identifier must be either an **in** or **in out** event port identifier in the interface namespace of the associated component type or an **out** or **in out** event port in the interface namespace of the component type associated with the named subcomponent.

Semantics

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 switching are discussed in Section 12.3.

A mode represents an operational state that is represented as a runtime configuration of containee components, connections, and mode-specific property value associations. A runtime configuration of interconnected systems, processes and threads is such an operational state. As is a collection of execution platform components. An operational mode embedded in the source text may be represented as threads and data with modes and different associated property values.

Systems and their components may have mode-specific property value associations. The modes for subcomponents, connections, flow implementations or property associations are specified using the associated **in modes** statement.

A component implementation may contain several declared modes. Exactly one of those modes is the *current* mode. Initially, the initial mode is the current mode.

In the case of modes declared in system and process implementations, 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 terminated state.

In the case of modes in threads, the current mode reflects conditional execution within the source text. Observable differences in execution can be reflected in AADL by mode-specific call sequences, flow implementations, connections, and property associations (see Section 10.4).

In the case of execution platforms, only the execution platform components that are part of the current mode are accessible to software components. Only the processors and memories that are part of the current mode can be the target of bindings of components active in that mode.

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 implementation. 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. The reserved word **none** is used to indicate a declaration is not part of any mode.

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. If a connection 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. The reserved word **none** is used to indicate a declaration is not part of any mode or mode transition.

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 Switch

The modes subclause declares a state machine describing the dynamic mode switching behavior of modes. The states of the state machine represent the different modes and the transitions specify the event(s) that can trigger a mode switch to the destination mode. Only one mode alternative represents the current mode at any one time. A mode switch is triggered when an event arrives at an event port that is named in one of the transitions out of the state representing the current mode. If an event is raised and there is no transition out of the current mode naming the event port through which the event arrives, the event is ignored. If several 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 mode transition, then the mode transition with the highest urgency takes precedence.

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.

A mode switch within a thread results in a change of its current mode. The effect is a change in the subprogram call sequence and mode-specific property values to reflect a change in source text internal execution behavior. Such a change in property values may include a change in the thread's period, deadline, or worst-case execution time. A mode switch within a thread does not affect the set of active threads, processors, devices, buses, or memories, nor does it affect the set of active connections.

A mode switch 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. 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.

When a mode switch is requested through the arrival of an event on a mode transition, the actual mode switch occurs immediately if no periodic threads are part of the old mode, otherwise it occurs once these periodic threads in the old mode are synchronized at their hyperperiod. Only those threads with a `Synchronzied_Component` property value of true are considered in the determination of the hyperperiod (see Section 12.3).

Starting with the actual time of mode switch, the component is in a *mode transition in progress state* for a limited amount of time. During this time some threads are deactivated, other threads are activated, connections are adjusted, and the active threads in the new mode start to execute. This time period takes the `Synchronized_Component` property into account and is determined at the level of the whole system instance (see Section 12.3). After that period of time, the component is considered to operate in the new mode.

At the time of the actual mode switch, the deactivate entrypoint is invoked for the following threads that must be deactivated: periodic threads that are synchronized with the mode switch; aperiodic or sporadic threads that are in the suspended awaiting dispatch state. This is shown in Figure 5 with the transition labeled thread **exit(mode)**.

At the time instant of actual mode switch, aperiodic and sporadic threads as well as periodic threads not synchronized with the mode switch may still be in the perform computation state (see Figure 5). The `Active_Thread_Handling_Protocol` property specifies for each such thread what action is to be taken. Possible actions are:

Abort the execution of the thread and permit the thread to recover any state through execution of its recover entrypoint. This permits the thread to recover to a consistent state for future activation and dispatch. Upon completion of the recover entrypoint, execution, event and event data port queues of the thread are flushed and the thread enters the suspended awaiting mode state. If the thread was executing a server subprogram, the current dispatch execution of the calling thread of a call in progress or queued call is also aborted.

Permit the thread to complete the execution of its current dispatch. 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.

Permit the thread to finish processing all events or event data in its queues.

The semantics of any such actions for threads in the performing computation state at the time instant of actual mode switch is not shown in the hybrid automaton in Figure 5.

Background processes that are only part of the old mode are suspended when the actual mode switch occurs.

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. In the case of periodic threads, this is immediately followed by their first dispatch of the compute entrypoint. In the case of background threads, the thread resumes execution from where it was suspended at the last deactivation.

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.

Some property values for a component or its subcomponents may be mode-specific, for example the period of a periodically dispatched thread may be different in different modes of operation. It changes at the time of actual mode switch.

Processing Permissions and Requirements

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.

If two different events that occur logically simultaneously result in more than one possible transition out of the current mode, a method of implementation may supply an implementation-dependent order or response to these events. An implementation may provide an `Urgency` parameter to the `Raise_Event` service call (see Section 5.3) to prioritize the response to simultaneous events. A method of implementation is permitted to raise a runtime error to indicate the nondeterministic nature of the system. Or, a method of implementation may specify additional rules to define the order in which transitions will occur.

In a physical distributed system, exact simultaneity among multiple events cannot be achieved. A physical system implementation must use synchronization protocols sufficient to insure that the causal ordering of event and data transfers defined by the logical temporal semantics of this standard are satisfied by the physical system, to the degree of assurance required by an application.

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`.

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.

Examples

```
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 GPD_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
   data port Main_Gps.Position -> Position in modes (Dualmode, Mainmode);
   data port Backup_Gps.Position -> Position in modes (Backupmode);
   data port Backup_Gps.Position -> Main_Gps.SecondaryPosition
                                             in modes (Dualmode);
modes
   Initialize: initial mode;
   Dualmode : mode;
   Mainmode : mode;
   Backupmode: mode;
   Initialize -[ Init_Done ]-> Dualmode;
   Dualmode -[ Monitor.Backup_Stopped ]-> Mainmode;
   Dualmode -[ Monitor.Main_Stopped ]-> Backupmode;
   Mainmode, 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
   data 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;
```

12 Operational System

Component type and component implementation declarations are architecture design elements that define the structure and connectivity of a physical 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 physical 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.

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.

12.1 System Instances

A system instance represents the runtime architecture of a physical system that consists of application software components and execution platform components.

A system instance is *completely instantiable* if the system implementation being instantiated is completely specified and completely resolved.

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 server subprograms as necessary.

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.

Legality Rules

The system type specified for a system instance must not contain any required subcomponents.

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.

In a complete system instance, the ports of all threads, devices, and processors must be the ultimate source or destination of semantic connections. The `Required_Connection` property may be used to indicate that a port connection is optional. In the case of the predeclared `Error` and `Complete` ports (see Section 5.3), connections are optional.

In a completely instantiable system, the subprogram calls of all threads must either be local calls or be bound to a server subprogram whose thread is part of the same mode.

In a completely instantiable system, 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.

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 or their predeclared `Dispatch` port. The predeclared `Dispatch` port must not be connected if the thread has a `Dispatch_Protocol` property value of `periodic` or `background`.

For instantiable systems, all threads must be bindable to processors and all components representing source text must be bindable to memory.

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.

A 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.

Semantics

A system instance represents an operational physical system. That physical system may be a stand-alone system or a system of systems. A system instance consists of application software 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.

The physical 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 processed to generate binary images. The binary images are loaded into memory and made accessible to threads in virtual address spaces of processes.

In addition, there exists a kernel address space for every processor. This address space contains binary images of processor software and device software bound to the processor.

12.2 System Binding

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 physical system implementation are collectively called bindings.

Naming Rules

The `Allowed_Processor_Binding` property values evaluate to a processor, or a system that contains a processor in its component containment hierarchy.

The `Allowed_Memory_Binding` property values 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.

The first identifier of the property value for each element of a property value list must exist in the local namespace of the containing component implementation or in one of its containing component implementations. The first containing component with a match must be an execution platform system, i.e., must be a processor component or a system component containing a processor or memory in its containment sub-hierarchy. Subsequent identifiers must exist in the local namespace of the component implementation associated with the component identified by the preceding identifier. The final identifier identifies the system, processor, or memory component that represents a legal candidate for processor or memory binding.

Legality Rules

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). In the case of dynamic process loading, the actual binding may change at runtime. In the case of tightly coupled multi-processor configurations, 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. Multiple software components may be bound to a single memory component. A software component may be bound to multiple memory components. A thread must be bound to a single processor. Multiple threads can be bound to a single processor.

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.

A shared data component must be bound to memory accessible by all processors to which the processes sharing the data component are bound.

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.

The complete set of software components making up the kernel address space of a processor must be bound to memory that is accessible by that processor.

Each thread must be bound to a processor satisfying the `Allowed_Processor_Binding_Class` and `Allowed_Processor_Binding` property values of the thread. 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.

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.

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 insure 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.

The memory requirements of ports and data components are specified as property values of their data types. Those property associations can have binding-specific values.

Semantics

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 with in the system iimplementation.

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

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 physical 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 physical system construction, and may require thread to processor bindings to be fixed at mode changes. Other choices are possible and permitted.

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

```
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;
```

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.

12.3 System Operation

System operation is the execution of a completely instantiated and bound system. System operation consists of different phases:

System startup: initialization of the execution platform and the application system.

Normal operation: execution of threads and communication between threads and devices.

System operation mode transition: mode switching of one or more components with specified mode transitions.

System-wide fault handling, shutdown, and restart.

System Startup

On system startup, the hardware of the execution platform is initialized, 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. Loading into memory may take zero time, if the memory can be preloaded, e.g., PROM or flash memory. Once initialized, each processor initiates the loading of the binary images of processes bound to the specific processor into memory (see Figure 17).

Process binary images are loaded in the memory component to which the process and its contained software components are bound (see Figure 8). 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.

The maximum system initialization time can be determined as `Processor_Startup_Deadline` + max(`Load_Time`) of all systems and processes + max(`Initialize_Deadline`) of all threads.

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.

Data components shared across processes must be bound to memory accessible by all processors to which the processes sharing the data component are bound.

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.

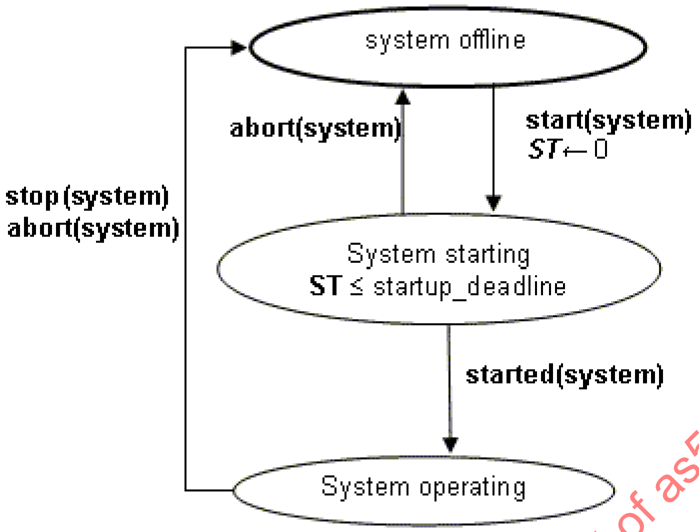


Figure 17 System Instance States, Transitions, and Actions

Normal System Operation

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.

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.

This version of the standard defines the semantics of execution for synchronous systems. A method of implementing a system may provide asynchronous system semantics as long as system wide coordination protocols are in place.

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.

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.

System Operation Modes

The set of all mode transitions specified for all components of a system instance form a set of concurrent mode transitions, called *system operation modes* (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.

The discrete variable **Mode** denotes a SOM. That is, the variable **Mode** denotes a possible discrete state that is defined by the mode hybrid semantic diagrams. 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.

The SOM transition is requested whenever a mode transition in any component in the system instance is requested by the arrival of an event. A single event can trigger a mode switch request in one or more components. In a synchronized system, this event occurs logically simultaneously for all components, i.e., the resulting component mode switch requests are treated as a single SOM transition request.

If several events occur logically simultaneously and 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 considered to have an implementation-dependent order that determines the mode transition for the mode switch – resulting in the other events being ignored.

After a SOM transition request has occurred, the actual SOM transition occurs in zero time, if no periodic threads are part of the old mode, otherwise, it occurs at the hyperperiod boundary of the old SOM. This is indicated in Figure 18 by the guard on the transition from the `current_system_operation_mode` state to the `mode_transition_in_progress` state. During that time, the system continues to operate in the old SOM and additional events that would result in a SOM transition from the current SOM are ignored.

The rational-valued function Hyper(**Mode**) in Figure 18 denotes the hyperperiod of a SOM. The hyperperiod is determined by the periods of those periodic threads whose `Synchronzied_Component` property is true, and that will deactivate or activate as part of the mode switch, or that remain active but whose connections may change during the mode switch. If this set of threads is empty, the mode transition is initiated immediately.

At the time of actual SOM transition, the transition is performed to the new SOM that contains the destination modes of the requested component mode switch(es).

System Operation Mode Transition

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 arriving events that appear in transition edge declarations are ignored and will not cause any mode change.

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. For data connections, this means that the data value is not transferred into the **in** data port variable of the newly disabled thread.

At the instant of time the `mode-transition-in-progress` state is entered, data is transferred logically simultaneously for all connections that are declared to be part of any of the component mode transitions making up the SOM transition. For data connections, this means that the data is transferred from the **out** data port such that its value becomes available at the first dispatch of the receiving thread.

At the instant of time the `mode-transition-in-progress state` is entered, connections that are not part of the old SOM and part of the new SOM are enabled. For data connections, this means that the data value of a transition connection is transferred into the **in** data port variable of the newly enabled thread. If the **in** data port of the destination thread is not the destination of a transition connection, 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. If the source thread is also activated as part of the mode transition, its **out** data port value is transferred after the thread completes its activate entrypoint execution.

When the mode-transition-in-progress state is entered, *thread* **exit(Mode)** is triggered for all threads that are part of the old thread and not part of the new thread. This results in the execution of deactivation entrypoints for those threads (see Figure 5) as described in Section 11.

In addition, at the time the `mode-transition-in-progress` state is entered, *thread* **enter(Mode)** is triggered for threads that are part of the new mode and not part of the old mode. This permits those threads to execute their activation entrypoints (see Figure 5). In addition, for periodic threads this is immediately followed by their first compute entrypoint dispatch as described in Section 11.

At the instant of time the `mode-transition-in-progress` state is entered, connections that are not part of the old SOM and are part of the new SOM are enabled, i.e., connection transmission occurs according to the connections that are part of the new SOM.

While the system is in the `mode-transition-in-progress` state, threads that are part of the old and new SOM continue to operate normally. SOM transition requests as resulting from raise events are ignored while the system instance is in the `mode-transition-in-progress` state.

The system instance remains in the `mode-transition-in-progress` state until the next hyperperiod. This hyperperiod is determined by the rules stated earlier. At that time, the system instance enters `current_system_operation_mode` state and starts responding to new requests for SOM transition.

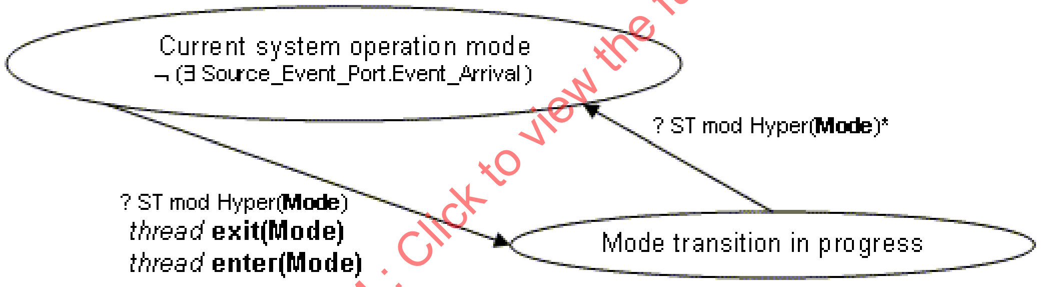

Current system operation mode
¬ (∃ Source_Event_Port.Event_Arrival)
? ST mod Hyper(Mode)*
? ST mod Hyper(Mode)
thread exit(Mode)
thread enter(Mode)
Mode transition in progress

Figure 18 System Mode Switch Semantics

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.

System-wide Fault Handling, Shutdown, and Restart

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.

Processing Requirements and Permissions

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 physical system from a specification.

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.

A method of building systems is permitted to only support static process loading.

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.

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.

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.

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.

A method of building systems is permitted to omit loading of processor, device, and bus software in a processor kernel address space if none of the threads bound to that processor need to access or execute that software.

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.

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.

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.

13 Lexical Elements

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.

13.1 Character Set

The only characters allowed outside of comments are the graphic_characters and format_effectors.

Syntax

```
character ::= graphic_character | format_effector
            | other_control_function


graphic_character ::= identifier_letter | digit | space_character
                    | special_character
```

Semantics

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).

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.

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_function

Any control function, other than a format_effector, that is allowed in a comment; the set of other_control_functions allowed in comments is implementation defined.

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

In a nonstandard mode, the implementation may support a different character repertoire; in particular, the set of characters 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.

13.2 Lexical Elements, Separators, and Delimiters

Semantics

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.

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.

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.

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.

A *delimiter* is either one of the following special characters

( ) [ ] { } , . : ; = * + -

or one of the following *compound delimiters* each composed of two or three adjacent special characters

:: => +=> -> ->> .. –[ ]–> { **}**

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.

The following names are used when referring to compound delimiters:

Delimiter	**Name**
::	qualified name separator
=>	association
+=>	additive association
->	Immediate connection
->>	delayed connection
..	interval
-[	left step bracket
]->	right step bracket
{**	begin annex
**}	end annex

Processing Requirements and Permissions

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.

13.3 Identifiers

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.

Semantics

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.

Legality Rules

An identifier must be distinct from the reserved words of the AADL.

Processing Requirements and Permissions

In a nonstandard mode, an implementation may support other upper/lower case equivalence rules for identifiers, to accommodate local conventions.

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
```

13.4 Numerical Literals

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
```

13.4.1 Decimal Literals

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 ]
decimal_integer_literal ::= numeral
numeral ::= digit {[underline] digit}*
exponent ::= E [+] numeral | E – numeral
positive_exponent ::= E [+] numeral
```

Semantics

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.

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
```

13.4.2 Based Literals

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

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

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.

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#240#      -- integer literals of value 224
```

13.5 String Literals

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
```

A string_element is either a pair of quotation marks (""), or a single graphic_character other than a quotation mark.

Semantics

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.

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"
```

13.6 Comments

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

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
```

13.7 Reserved Words

The following are the AADL reserved words. Reserved words are case insensitive.

aadlboolean	**aadlinteger**	**aadlreal**	**aadlstring**
access	**all**	**and**	**annex**
applies	**binding**	**bus**	**calls**
classifier	**connections**	**constant**	**data**
delta	**device**	**end**	**enumeration**
event	**extends**	**false**	**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**	**provides**
public	**range**	**reference**	**refined**
refines	**requires**	**server**	**set**
sink	**source**	**subcomponents**	**subprogram**
system	**thread**	**to**	**true**
type	**units**	**value**	

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.

Appendix A Predeclared Property Sets

Normative

The property set `AADL_Properties` is a part of every AADL specification. It defines properties for AADL model elements that are defined in the core of the AADL. This property set may not be modified by the modeler.

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.

The property types, property names, and property constants of these predeclared property sets can be named without property set name qualification.

A.1 Standard AADL Property Set

There is a standard predeclared property set named `AADL_Properties`. This property set declaration is a part of every AADL specification.

NOTES:

In accordance with the naming rules for references to items defined in the predeclared property sets, the declarations in this property set refer to enumeration types and property constants declared in the AADL_Project property set without a qualifying property set name.

property_set `AADL_Properties` **is**

`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.

`Activate_Entrypoint:` **aadlstring**

  applies to (**thread**);

    The `Activate_Entrypoint` property specifies the name of a source text code sequence that will execute when a thread is activated. This property may have an unspecified value.

The named code sequence in the source text must be visible in and callable from the outermost program scope, as defined by the scope and visibility rules of the applicable source language. The source language annex of this standard defines acceptable parameter and result signatures for the entrypoint subprogram.

```
Active_Thread_Handling_Protocol:
   inherit Supported_Active_Thread_Handling_Protocols
         => value(Default_Active_Thread_Handling_Protocol)

   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 implementer defined. One of the available choices must be the default value.

This protocol specifies the activation order of threads become active as part of a new mode.

```
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 deactiveated until it is reactivated.

```
Actual_Connection_Binding: inherit reference (bus, processor, device)

   applies to (port connections, thread, thread group, process, system);
```

Connections are bound to the bus, processor, or device specified by the `Actual_Connection_Binding` property.

```
Actual_Latency: Time

   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`. The standard units are ns (nanoseconds), us (microseconds), ms (milliseconds), sec (seconds), min (minutes) and hr (hours).

```
Actual_Memory_Binding: inherit reference (memory)

   applies to (thread, thread group, process, system, processor,
               data port, event data port, subprogram);
```

Code and data from source text is bound to the memory specified by the `Actual_Memory_Binding` property.

`Actual_Processor_Binding:` **`inherit reference`** `(`**`processor`**`)`

`applies to` `(`**`thread`**`,` **`thread group`**`,` **`process`**`,` **`system`**`);`

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.

`Actual_Subprogram_Call:` **`reference`** `(`**`server subprogram`**`)`

`applies to` `(`**`subprogram`**`);`

The `Actual_Subprogram_Call` property specifies the server subprogram that is servicing the subprogram call as a remote call. If no value is specified, the subprogram call is a local call.

`Actual_Subprogram_Call_Binding:` **`reference`** `(`**`bus`**`,` **`processor, memory`**`)`

`applies to` `(`**`subprogram`**`);`

The `Actual_Subprogram_Call_Binding` property specifies the bus, processor, or memory to which a remote subprogram call is bound. If no value is specified, the subprogram call is a local call.

`Actual_Throughput: Data_Volume`

`applies to` `(`**`flow`**`);`

The `Actual_Throughput` property specifies the actual throughput as determined by an analysis of the semantic flows representing the end-to-end flow.

`Aggregate_Data_Port:` **`aadlboolean`** `=>` **`false`**

`applies to` `(`**`port group`**`);`

The `Aggregate_Data_Port` property specifies whether the port group acts as an aggregate data port for ports that contain data.

```
Allowed_Access_Protocol: list of enumeration (Memory_Access,
                                              Device_Access)
```

`applies to` `(`**`bus`**`);`

The `Allowed_Access_Protocol` property specifies the categories of hardware components that can be connected to the processor via the bus. The `Memory_Access` value specifies that threads executing on a processor may access portions of their binary image that have been bound to a memory over that bus. The `Device_Access` value specifies that threads executing on the processor can communicate with devices via that bus.

If a list of allowed connection protocols is not specified for a bus, then the bus may be used to connect both devices and memory to the processor.

`Allowed_Connection_Binding:` **`inherit list of reference`** `(`**`bus, processor, device`**`)`

`applies to` `(`**`port connections`**`,` **`thread group`**`,` **`process`**`,` **`system`**`);`

The `Allowed_Connection_Binding` property specifies the execution platform resources that are to be used to perform a communication. The property type is a list of component names. The list must contain an odd number of component names. The named components must belong to a processor, device or bus category.

The first component named in the list must be either the processor to which the thread containing the ultimate source feature is bound, or else the processor or device containing the ultimate hardware source feature. The last component named in the list must be either the processor to which the thread containing the ultimate destination feature is bound, or else the processor or device containing the ultimate hardware destination feature. The intermediate component names must alternate between a bus, and a processor or a device. Each pair of names for processor or device components that are separated by the name of a bus component must share that bus component. That is, the sequence of processors, devices and buses must form a connected path through the specified hardware architecture.

```
Allowed_Connection_Binding_Class:
   inherit list of classifier(processor, bus, device)
   applies to (port connections, thread, thread group, process, system);
```

The `Allowed_Connection_Binding_Class` property specifies the hardware resources that are to be used to perform a communication. The property type is list of component classifier names. The list must contain an odd number of component names. The named component classifiers must belong to a processor, device or bus category.

The first component classifier named in the list must be either that of the processor to which the thread containing the ultimate source feature is bound, or else the processor or device containing the ultimate hardware source feature. The last component classifier named in the list must be either that of the processor to which the thread containing the ultimate destination feature is bound, or else the processor or device containing the ultimate hardware destination feature. The intermediate component classifier names must alternate between a bus, and a processor or a device. Each pair of names for processor or device classifiers that are separated by the name of a bus classifier must share that bus component. That is, the sequence of processors, devices, and buses must form a connected path through the specified hardware architecture.

```
Allowed_Connection_Protocol: list of enumeration
                                    (Data_Connection,
                                     Event_Connection)
    applies to (bus);
```

The `Allowed_Connection_Protocol` 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. The `Data_Connection` value means data connections are supported. The `Event_Connection` value means event connections are supported.

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_Memory_Binding: inherit list of reference (memory, system, processor)
   applies to (thread, thread group, process, system, device, 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.

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

   applies to (thread, thread group, process, system, device, 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_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.

```
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_Processor_Binding: **inherit list of reference** (**processor, system**)

applies to (**thread, thread group**, **process**, **system, device**);

The Allowed_Processor_Binding property specifies the set of processors that are available for binding. The set is specified by a list of 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 one of the specified set of 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 it the thread associated with the device driver code can be bound to oe 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.

Allowed_Processor_Binding_Class:
inherit list of classifier (**processor, system**)

applies to (**thread, thread group**, **process**, **system, device**);

The Allowed_Processor_Binding_Class property specifies a set of processor and system classifiers. These component classifiers constrain the set of processors in the Allowed_Processor_Binding property to the subset that satisfies the component classifier.

The default value is inherited from the containing process or system component.

If this property has no associated value, then all processors specified in the Allowed_Processor_Binding are acceptable candidates.

Allowed_Subprogram_Call: **list of reference** (**server subprogram**)

applies to (**subprogram**);

A subprogram call can be bound to any member of the set of server subprograms specified by the Allowed_Subprogram_Call property. If no value is specified, then subprogram call must be a local call.

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

applies to (**subprogram**, **thread**, **thread group**, **process, system**);

Server 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 must be a local call.

Assign_Time: Time

applies to (**processor**, **bus**);

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 or bus. The time required is assumed to be the number of bytes times the Assign_Byte_Time plus

Assign_Fixed_Time.

Assign_Time = (Number_of_Bytes * Assign_Byte_Time) + Assign_Fixed_Time

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.

Assign_Byte_Time: Time

applies to (**processor**, **bus**);

The Assign_Byte_Time property specifies a time unit value which reflects the time required to move a single bytes on a particular processor or bus, not including the Assign_Fixed_Time.

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.

Assign_Fixed_Time: Time

applies to (**processor**, **bus**);

The Assign_Fixed_Time property specifies a time unit value which reflects the fixed or overhead time required for assignment of any number of bytes on a particular processor or bus.

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.

Available_Memory_Binding: **inherit list of reference** (**memory, system**)

applies to (**system**);

The Available_Memory_Binding property specifies the set of contained memory components that are made available for binding outside the system. The set is specified by a list of memory and system component names. System names represent the memories contained in them.

Available_Processor_Binding: **inherit list of reference** (**processor, system**)

applies to (**system**);

The Available_Processor_Binding property specifies the set of contained processor components that are made available for binding outside the system. The set is specified by a list of processor and system component names. System names represent the processors contained in them.

Base_Address: **access aadlinteger** 0 **..** **value**(Max_Base_Address)

applies to (**data**);

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.

The property expression must be preceded by the reserved word **access**, in which case the property specifies the address associated with the access to a subcomponent rather than the data component itself.

`Client_Subprogram_Execution_Time: Time`

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`. 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_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.

`Compute_Deadline: Time`

applies to (**thread**, **subprogram**, **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 server subprogram, the `Compute_Deadline` applies to the thread executing the remote procedure call in response to the server 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).