



AEROSPACE STANDARD

AS5506™**REV. D**Issued 2009-01
Revised 2022-04

Superseding AS5506C

Architecture Analysis & Design Language (AADL)

RATIONALE

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.

The language was originally published as SAE AS5506 in 2004. The language has been refined and extended based on industrial experience as AADL V2 and published as AS5506A in 2009. The improvements focused on better support for architecture templates and modeling of layered and partitioned architectures. AADL V2.1, a revision that addresses a number of errata that have been reported and agreed upon by the committee, was published as AS5506B in 2012. A subsequent revision, AS5506C or AADL V2.2, has been published in 2017. This third revision addresses a number of additional errata and minor improvements to the language since AS5506C. These errata and changes have been approved by the committee.

SAE Executive Standards Committee Rules provide that: "This report is published by SAE to advance the state of technical and engineering sciences. The use of this report is entirely voluntary, and its applicability and suitability for any particular use, including any patent infringement arising therefrom, is the sole responsibility of the user."

SAE reviews each technical report at least every five years at which time it may be revised, reaffirmed, stabilized, or cancelled. SAE invites your written comments and suggestions.

Copyright © 2022 SAE International

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of SAE.

TO PLACE A DOCUMENT ORDER: Tel: 877-606-7323 (inside USA and Canada)
Tel: +1 724-776-4970 (outside USA)
Fax: 724-776-0790
Email: CustomerService@sae.org
<http://www.sae.org>

SAE WEB ADDRESS:

Full document details: SAE AS5506D

For more information on this standard, visit
<https://www.sae.org/standards/content/AS5506D/>

Documents

Document Sharing Hub

TABLE OF CONTENTS

1.	SCOPE	8
1.1	Purpose/Extent	8
1.2	Field of Application	9
1.3	Structure of Document	9
1.3.1	A Reader's Guide	9
1.3.2	Structure of Clauses and Subclauses	11
1.4	Error, Exception, Anomaly and Compliance	12
1.5	Method of Description and Syntax Notation	13
1.6	Method of Description for Discrete and Temporal Semantics	15
2	REFERENCES	19
2.1	Normative References	19
2.2	Informative References	19
2.3	Terms and Definitions	19
3	ARCHITECTURE ANALYSIS & DESIGN LANGUAGE SUMMARY	20
4	COMPONENTS, PACKAGES, AND ANNEXES	25
4.1	AADL Specifications	25
4.2	Packages	27
4.3	Component Types	31
4.4	Component Implementations	36
4.5	Subcomponents	42
4.6	Abstract Components	49
4.7	Prototypes	52
4.8	Annex Subclauses and Annex Libraries	57
5	SOFTWARE COMPONENTS	59
5.1	Data	59
5.1.1	Runtime Support For Shared Data Access	62
5.2	Subprograms and Subprogram Calls	65
5.3	Subprogram Groups and Subprogram Group Types	72
5.4	Threads	74
5.4.1	Thread States and Actions	78
5.4.2	Thread Dispatching	81
5.4.3	Thread Scheduling and Execution	84
5.4.4	Execution Fault Handling	85
5.4.5	Thread Internal Modes and Mode Transitions	87
5.4.6	System Synchronization Requirements	87
5.4.7	Asynchronous Systems	87
5.4.8	Runtime Support For Threads	88
5.5	Thread Groups	89
5.6	Processes	92
6	EXECUTION PLATFORM COMPONENTS	96
6.1	Processors	97
6.2	Virtual Processors	101
6.3	Memory	104
6.4	Buses	106
6.5	Virtual Buses	109
6.6	Devices	112
7	SYSTEM COMPOSITION	117
7.1	Systems	117
8	FEATURES AND SHARED ACCESS	120
8.1	Abstract Features	122



8.2	Feature Groups and Feature Group Types	123
8.3	Ports.....	128
8.3.1	Port Categories.....	130
8.3.2	Port Input and Output Timing	131
8.3.3	Port Queue Processing	133
8.3.4	Events and Subprograms.....	134
8.3.5	Runtime Support For Ports.....	135
8.4	Subprogram and Subprogram Group Access	138
8.5	Subprogram Parameters	143
8.6	Data Component Access	144
8.7	Bus Component Access	148
8.8	Virtual Bus Component Access.....	150
8.9	Internal Features and Processor Features.....	151
9	CONNECTIONS	152
9.1	Feature Connections	153
9.2	Port Connections	155
9.2.1	Port Connection Characteristics.....	160
9.2.2	Port Connection Topology.....	161
9.2.3	Connection Patterns for Component Arrays and Feature Arrays	162
9.2.4	Port Communication Timing	164
9.2.5	Sampled, Immediate, and Delayed Data Port Communication.....	164
9.2.6	Semantic Port Connections and Port Queues	166
9.3	Parameter Connections	168
9.4	Access Connections	170
9.5	Feature Group Connections	173
10	FLOWS.....	179
10.1	Flow Specifications	180
10.2	Flow Implementations	184
10.3	End-To-End Flows	188
11	PROPERTIES	192
11.1	Property Sets	192
11.1.1	Property Types	193
11.1.2	Property Definitions	197
11.1.3	Property Constants.....	199
11.2	Predeclared Property Sets.....	200
11.3	Property Associations	201
11.4	Property Expressions.....	207
12	MODES AND MODE TRANSITIONS	213
13	OPERATIONAL SYSTEM.....	221
13.1	System Instances.....	221
13.2	System Binding	222
13.3	System Startup.....	226
13.4	Normal System Operation.....	227
13.5	System Operation Modes	227
13.6	System Operation Mode Transitions.....	228
13.7	System-wide Fault Handling, Shutdown, and Restart	230
14	LAYERED SYSTEM ARCHITECTURES	233
15	LEXICAL ELEMENTS	235
15.1	Character Set	235
15.2	Lexical Elements, Separators, and Delimiters	236
15.3	Identifiers.....	237
15.4	Numerical Literals	238
15.4.1	Decimal Literals.....	238
15.4.2	Based Literals.....	238



15.5	String Literals	239
15.6	Comments	239
15.7	Reserved Words	240
A.1	Predeclared Deployment Properties	241
A.2	Predeclared Thread Properties	249
A.3	Predeclared Timing Properties	252
A.4	Predeclared Communication Properties	259
A.5	Predeclared Memory Properties	262
A.6	Predeclared Programming Properties	266
A.7	Predeclared Modeling Properties	273
A.8	Project-Specific Property Set	274
A.9	Predeclared Runtime Services	279
C.1	Constraints on Component Containment	289
C.2	AADL Core Language Syntax Rules	292
C.3	AADL Core Language Meta Model Element Identifiers	319
D.1	Scope	323
D.2	AADL Graphical Symbols	323
D.3	Implementation Suggestions	331
ANNEX DOCUMENT A	ARINC653	336
ANNEX DOCUMENT B	BEHAVIOR MODEL	336
ANNEX DOCUMENT C	CODE GENERATION	336
ANNEX DOCUMENT D	DATA MODELING	336
ANNEX DOCUMENT E	ERROR MODEL	336
ANNEX DOCUMENT F	MINI ANNEXES	336
Annex F.1	Data Sets	336

FOREWORD

- (1) The AADL standard was prepared by the SAE Avionics Systems Division (ASD) Embedded Computing Systems Committee (AS-2) Architecture Description Language (AS-2C) subcommittee.
- (2) This standard addresses the requirements defined in SAE ARD5296, Requirements for the Avionics Architecture Description Language¹.
- (3) The AADL standard consists of a core language standard that is defined in this document and a collection of standardized property sets and/or sublanguages that are defined in annex documents. The core language standard provides full support for modeling the application task and communication architecture, the hardware platform, and the physical environment of embedded software-intensive systems, including standardized predeclared properties to characterize task execution and communication timing, as well as deployment of the application on the hardware platform. The standardized extensions allow core AADL models to be annotated with information that is not represented by the core language to meet specific embedded system analysis needs such as security analysis, dependability analysis, and behavioral analysis, and support for automated generation and integration of systems.
- (4) The starting point for the AADL standard development was MetaH, an architecture description language and supporting toolset, developed at Honeywell Technology Laboratories under the Defense Advanced Research Research Projects Agency (DARPA) and Army Aviation and Missile Command (AMCOM) sponsorship.
- (5) The AADL standard has been designed to be compatible with real-time operating system standards such as POSIX and ARINC 653.
- (6) The AADL standard is aligned with Object Management Group (OMG) Unified Modeling Language (UML) and Modeling and Analysis of Real-Time Embedded systems (MARTE) through a standardized profile for AADL (OMG formal/19-04-01).
- (7) The AADL standard provides guidelines for users to transition between AADL models and program source text written in Ada (ISO/IEC 8652/2007 (E) Ed.3) and C (ISO/IEC 9899:1999) in a separate annex document.

¹ This was the original name of the SAE AADL.

INTRODUCTION

- (1) The SAE Architecture Analysis & Design Language (referred to in this document as AADL) is a textual and graphical language used to design and analyze the software and hardware architecture of performance-critical real-time systems. These are systems whose operation strongly depends on meeting non-functional system requirements such as reliability, availability, timing, responsiveness, throughput, safety, and security. AADL is used to describe the structure of such systems as an assembly of software components mapped onto an execution platform. It can be used to describe functional interfaces to components (such as data inputs and outputs) and performance-critical aspects of components (such as timing). AADL can also be used to describe how components interact, such as how data inputs and outputs are connected or how application software components are allocated to execution platform components. The language can also be used to describe the dynamic behavior of the runtime architecture by providing support to model operational modes and mode transitions. The language is designed to be extensible to accommodate analyses of the runtime architectures that the core language does not completely support. Extensions can take the form of new properties and analysis specific notations that can be associated with components and are standardized themselves.
- (2) AADL was developed to meet the special needs of performance-critical real-time systems, including embedded real-time systems such as avionics, automotive electronics, or robotics systems. The language can describe important performance-critical aspects such as timing requirements, fault and error behaviors, time and space partitioning, and safety and certification properties. Such a description allows a system designer to perform analyses of the composed components and systems such as system schedulability, sizing analysis, and safety analysis. From these analyses, the designer can evaluate architectural tradeoffs and changes.
- (3) Since AADL supports multiple and extensible analysis approaches, it provides the ability to analyze the cross cutting impacts of change in the architecture in one specification using a variety of analysis tools. AADL is designed to be used with analysis tools that support the automatic generation of the source code needed to integrate the system components and build a system executive. Since the models and the architecture specification drive the design and implementation, they can be maintained to permit model driven architecture based changes throughout the system lifecycle.

INFORMATION AND FEEDBACK

- (1) The website at <http://www.aadl.info> is an information source regarding the SAE AADL standard. It makes available papers on AADL, its benefits, and its use.
- (2) The website provides links to SAE AADL related discussion forums.
- (3) The website provides information and a download site for AADL toolsets. It also provides links to other resources regarding the AADL standard and its use.
- (4) Questions and inquiries regarding working versions of annexes and future versions of the standard can be addressed to info@aadl.info.
- (5) Informal comments on this standard may be sent to the committee, as listed in aadl.info website.

1. SCOPE

- (1) This standard defines a language for describing both the software architecture and the execution platform architectures of performance-critical, embedded, real-time systems; the language is known as the SAE AADL. An AADL model describes a system as a hierarchy of components with their interfaces and their interconnections. Properties are associated to these constructions. AADL components fall into two major categories: those that represent the physical hardware and those representing the application software. The former is typified by processors, buses, memory, and devices, the latter by application software functions, data, threads, and processes. The model describes how these components interact and are integrated to form complete systems. It describes both functional interfaces and aspects critical for performance of individual components and assemblies of components. The changes to the runtime architecture are modeled as operational modes and mode transitions.
- (2) The language is applicable to systems that are:
 - real-time,
 - resource-constrained,
 - safety-critical systems,
 - and those that may include specialized device hardware.
- (3) This standard defines the core AADL that is designed to be extensible. While the core language provides a number of modeling concepts with precise semantics including the mapping to execution platforms and the specification of execution time behavior, it is not possible to foresee all possible architecture analyses. Extensions to accommodate new analyses and unique hardware attributes take the form of new properties and analysis specific notations that can be associated with components. Users or tool vendors may define these extensions. Extensions may be proposed as annex documents for inclusion in the AADL standard.
- (4) This standard does not specify how the detailed design or implementation details of software and hardware components are to be specified. Those details can be specified by a variety of software programming and hardware description languages. The standard specifies relevant characteristics of the detailed design and implementation descriptions, such as source text written in a programming language or hardware description language, from an external (black box) perspective. These relevant characteristics are specified as AADL component properties, and as rules of conformance between the properties and the described components.
- (5) This standard does not prescribe any particular system integration technologies, such as operating system or middleware application program interfaces or bus technologies or topologies. However, specific system architecture topologies, such as the ARINC 653 executives, can be modeled through software and execution platform components. AADL can be used to describe a variety of hardware architectures and software infrastructures. Integration technologies can be used to implement a specified system. The standard specifies rules of conformance between AADL system architecture specifications and actual system implementations.
- (6) The standard was not designed around a particular set of tools. It is anticipated that systems and software tools will be provided to support the use of AADL.

1.1 Purpose/Extent

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

- (2) AADL describes application software and execution platform components of a system, and the way in which components are assembled to form a complete system or subsystem. The language addresses the needs of system developers in that it can describe common functional (control and data flow) interfacing idioms as well as performance-critical aspects relating to timing, resource allocation, fault-tolerance, safety and certification.
- (3) AADL describes functional interfaces and non-functional properties of application software and execution platform components. The language is not suited for detailed design or implementation of components. AADL may be used in conjunction with existing standard languages in these areas. AADL describes interfaces and properties of execution platform components including processor, memory, communication channels, and devices interfacing with the external environment. Detailed designs for such hardware components may be specified by associating source text written in a hardware description language.. 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, or domain-specific modeling languages.
- (4) AADL describes how components are composed together and how they interact to form complete system architectures. Runtime semantics of these components are specified in this standard. Various mechanisms are available to exchange control and data between components, including message passing, event passing, synchronized access to shared components, and remote procedure calls. Thread scheduling protocols and timing requirements may be specified. Dynamic reconfiguration of the runtime architecture may be specified through operational modes and mode transitions. The language does not require the use of any specific hardware architecture or any specific runtime software infrastructure.
- (5) Rules of conformance are specified between specifications written in AADL, source text and physical components described by those specifications, and physical systems constructed from those specifications. The AADL is not intended to describe all possible aspects of any possible component or system; selected syntactic and semantic requirements are imposed on components and systems. Many of the attributes of an AADL component are represented in an AADL model as properties of that component. The conformance rules of the language include the characteristics described by these properties as well as the syntactic and semantic requirements imposed on components and systems. Compliance between AADL specifications and items described by specifications is determined through analysis, e.g., by tools for source text processing and system integration.
- (6) AADL can be used for multiple activities in multiple development phases, beginning with preliminary system design. The language can be used by multiple tools to automate various levels of modeling, analysis, implementation, integration, verification and certification.

1.2 Field of Application

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

1.3 Structure of Document

1.3.1 A Reader's Guide

- (1) As necessary, the term AADL V2 will be used to refer to the revised version of AADL defined in this document.
- (2) The AADL standard consists of this core language document and a set of annex documents of standardized extensions. This core language document contains a number of sections and appendices. The sections define the core AADL. The appendices provide additional information, both normative and informative about the core language. Annex documents introduce additional standardized properties and possibly language extensions in the form of specialized notations.
- (3) AADL concepts are introduced in Section 3, Architecture Analysis & Design Language Summary. They are defined with full syntactic and semantic descriptions as well as naming and legality rules in succeeding sections. The

vocabulary and symbols of AADL are defined in Section 15. Appendix B , Glossary, provides informative definitions of terms used in this document. Other appendices include a syntaX SUMMARY and Predeclared Property Sets. The remainder of this section introduces notations used in this document and discusses standard conformance.

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

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

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

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

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

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

- Appendix A Predeclared Property Sets, contains the standard AADL set of predeclared properties, property types, and property constants.
- Appendix B Glossary, contains a glossary of terms.
- Appendix C Syntax Summary, contains a summary of the syntax as defined in the sections of this document.
- Appendix D Graphical AADL Notation, defines a graphical representation of AADL.
- Appendix E AADL Meta Model And XML Specification, defines an XML-based interchange format in form of an XML meta model and an XML schema. This appendix is deprecated.
- Appendix F Unified Modeling Language (UML) Profile, defines a profile for UML that extends and tailors UML to support modeling in terms of AADL concepts. This profile is defined in the context of the Object Management Group (OMG) Modeling and Analysis of Real-Time Embedded systems (MARTE).
- Appendix G Profiles And Extensions, contains profiles and extensions that have been approved by the standards body.

- (8) The annex documents introduce additions and extensions to the core AADL.
- Annex Document A ARINC653 provides guidance for modeling ARINC653 systems.
 - Annex Document B Behavior Modeling defines a standardized core language extension in the form of a sublanguage notation to specify the behavior of AADL components as AADL model annotations.
 - Annex Document C Code Generation discusses automatic generation and integration of runtime systems and application code in different implementation languages. It defines a standardized set of properties for recording mappings from the AADL model to source text and for automatic code generation.
 - Annex Document D Data Modeling provides guidance on data modeling and how to map relevant data modeling information into an AADL model if desirable. It defines a standardized set of properties and basic data types in support of data modeling.
 - Annex Document E Error Modeling, defines a standardized core language extension in the form of a sublanguage notation and properties the component to support annotating AADL models with safety-criticality and dependability related information of a system.
 - Annex Document F FACE provides a mapping with the FACE™ Technical Standard Edition 3.0.
- (9) The core language and the annex documents are *normative*, except for the material in each of the items listed below is informative:
- Text under a NOTE or Examples heading.
 - Each clause or subclause whose title starts with the word "Example" or "Examples".
- (10) All implementations shall conform to the core language. In addition, an implementation may conform separately to one or more Annexes that represent extensions to the core language.
- (11) The following appendices and annexes are informative and do not form a part of the formal specification of AADL:
- Appendix B Glossary
 - Appendix C Syntax Summary
 - Appendix G Profiles And Extensions

1.3.2 Structure of Clauses and Subclauses

- (1) Each section of the core standard is divided into clauses and subclauses that have a common structure. Each section, clause, and subclause first introduces its subject and then presents the remaining text in the following format. Not all headings are required in a particular clause or subclause. Headings will be centered and formatted as shown below.
- (2) All paragraphs are numbered with numbering restarting with each section. Naming rules, legality rules, and consistency rules have their own paragraph numbering also restarting with each section. They can be identified by section number and paragraph number.

Syntax

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

Naming Rules

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

Legality Rules

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

Consistency Rules

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

Standard Properties

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

Semantics

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

Runtime Support

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

Processing Requirements and Permissions

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

NOTES:

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

Examples

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

1.4 Error, Exception, Anomaly and Compliance

- (1) AADL can be used to specify dependable systems. A system can be compliant with its specification and this standard even when that system contains failed components that no longer satisfy their specifications. This section defines the terms fault, error, exception, anomaly and noncompliance [ISO/IEC/IEEE 24765 2010]; and defines how those terms apply to AADL specifications, physical components (implementations), models of components, and tools that accept AADL specifications as inputs.
- (2) A *fault* is defined to be an anomalous undesired change in execution behavior, possibly resulting from an anomalous undesired change in data being accessed by a thread or from violation of a compute time or deadline constraint. A fault in a physical component is a root cause that may eventually lead to a component error or failure. A fault is often a specific event such as a transistor burning out or a programmer making a coding mistake.
- (3) An *error* in a physical component occurs when an existing fault causes the internal state of the component to deviate from its nominal or desired operation. For example, a component error may occur when an add instruction produces an incorrect result because a transistor in the adding circuitry is faulty.
- (4) A *failure* in a physical component occurs when an error manifests itself at the component interface. A component fails when it does not perform its nominal function for the other parts of the system that depend on that component for their nominal operation.

- (5) A component failure may be a fault within a system that contains that component. Thus, the sequence of fault, error, failure may repeat itself within a hierarchically structured system. *Error propagation* occurs when a failed component causes the containing system or another dependent component to become erroneous.
- (6) A component may persist in a faulty state for some period of time before an error occurs. This is called *fault latency*. A component may persist in an erroneous state for some period of time before a failure occurs. This is called *error latency*.
- (7) An *exception* represents a kind of exceptional situation; it may occur for an erroneous or failed component when that error or failure is detected, either by the component itself or another component with which it interfaces. For example, a fault in a software component that eventually results in a divide-by-zero may be detected by the processor component on which it depends. An exception is always associated with a specific component. This document defines a standard model for exceptions for certain kinds of components (e.g., defines standard recovery sequences and standard exception events).
- (8) An *anomaly* occurs when a component is in an erroneous or failed state that does not result in a detect error. Undetected errors may occur in systems. A detected error may be handled using mechanisms other than the standard exception mechanisms. For example, an error may propagate to multiple components before it is detected and mitigated. This standard defines nominal and exceptional behaviors for components. Anomalies are any other undefined erroneous component behaviors, which are nevertheless considered compliant with this standard.
- (9) An AADL specification is *compliant* with the AADL core language standard if it satisfies all the syntactic and legality rules defined in Sections 4 - 15. An AADL specification is compliant with an AADL Annex standard if it satisfies all the syntactic and legality rules defined in the corresponding Annex.
- (10) A component or system is *compliant* with an AADL specification of that component or system if the nominal and exceptional behaviors of that component or system satisfy the applicable semantics of the AADL specification, as defined by the semantic rules in this standard. A component or system may be a physical implementation (e.g., a piece of hardware), or may be a model (e.g., a simulation or analytic model). A model component or system may exhibit only partial semantics (e.g., a schedulability model only exhibits temporal semantics). Physical components and systems must exhibit all specified semantics, except as permitted by this standard.
- (11) *Noncompliance* of a component with its specification is a kind of design fault. This may be handled by run-time fault-tolerance in an implemented actual system. A developer is permitted to classify such components as anomalous rather than noncompliant.
- (12) A tool that operates on AADL specifications is *compliant* with the core language standard if the tool checks for compliance of input specifications with the syntactic and legality rules defined herein, except where explicit permission is given to omit a check; and if all physical or model components or systems generated by the tool are compliant with the specifications used to generate those components or systems. The AADL standard allows profiles of language subsets to be defined and requires a minimum subset of the language to be supported (see Appendix G). A tool must clearly specify any portion of the language not supported and warn the user if a specification contains unsupported language constructs, when appropriate. A tool is compliant with a language extension annex if the tool checks for compliance of input specifications with the syntactic and legality rules defined in the respective annex document.
- (13) Compliance of an AADL specification with the syntactic and legality rules can be automatically checked, with the exception of a few legality rules that are not in general tractably checkable for all specifications. Compliance of a component or system with its specification, and compliance of a tool with this standard, cannot in general be fully automatically checked. A verification process that assures compliance to the degree required for a particular purpose must be used to perform the latter two kinds of compliance checking.

1.5 Method of Description and Syntax Notation

- (1) The language is described by means of a context-free syntax together with context-dependent requirements expressed by narrative rules. The meaning of a construct in the language is defined by means of narrative rules.
- (2) The context-free syntax of the language is described using the variant Backus-Naur Form (BNF) 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:

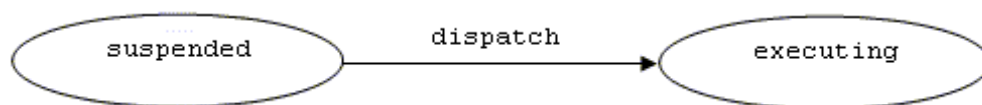
```
My_GPS: thread GPS.dualmode ;
```

1.6 Method of Description for Discrete and Temporal Semantics

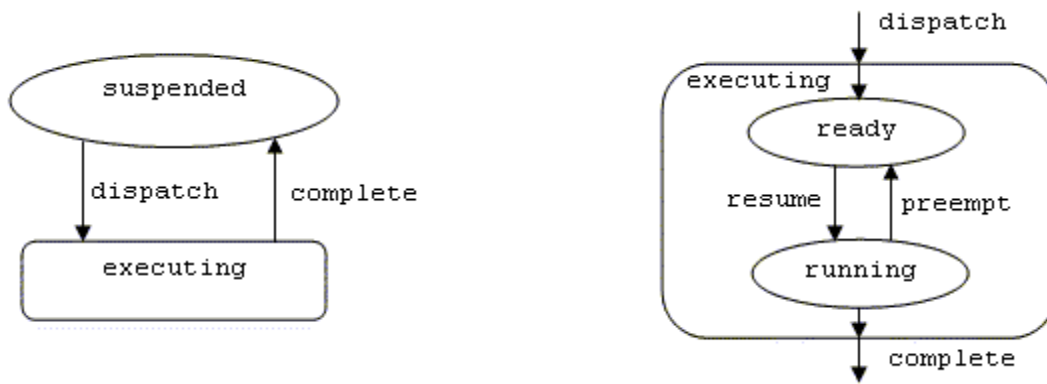
- Discrete and temporal semantics of the language are defined in sections that define AADL concepts using a concurrent hierarchical hybrid automata notation presented in [Henz 96], together with additional narrative rules about those diagrams. This notation consists of a hierarchical finite state machine 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 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,



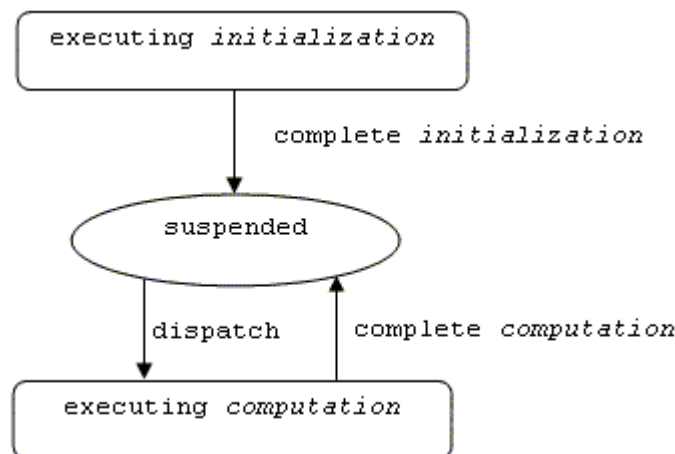
- Directed edges labeled with one or more comma-separated, lower case phrases are used to denote possible transitions between the discrete states of a component. Transitions over an edge are logically instantaneous, i.e., the time interval in which a transition from a discrete state (called the source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



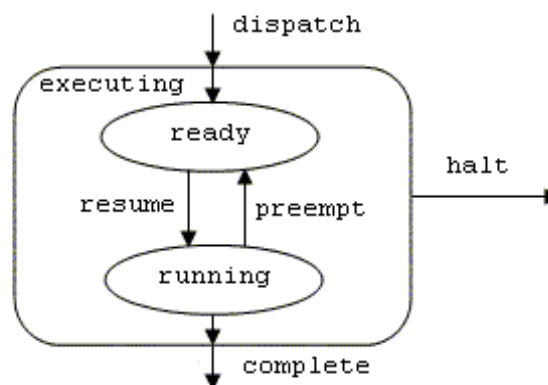
- Permissions that allow a runtime implementation of a transition to occur over an interval of time are expressed as narrative rules. However, all implemented transitions must be atomic with respect to each other, all observable serializations must be admitted by the logical semantics, and all temporal predicates as defined in subsequent paragraphs must be satisfied.
- Hybrid automaton can have hierarchical states. Oblong boxes labeled with lower case phrases denote abstract discrete states, for which another hybrid semantics diagram with an identically labeled oblong box for which another hybrid semantics diagram with an identically labeled oblong box specifies the discrete states and edges that make up that abstract discrete state. For example,



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



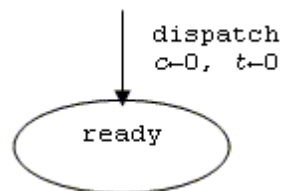
- (9) If there is an external edge that enters or exits an abstract discrete state in the defining diagram for, and there are no edges within that definition that connect any internal discrete state with that external edge, then there implicitly exist edges from every contained discrete state in the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



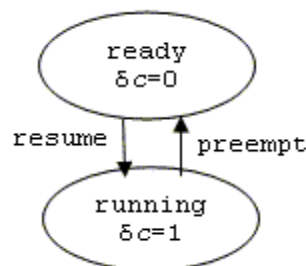
- (10) Real-valued variables whose values are time-varying may appear in expressions that annotate discrete states and edges of hybrid semantic diagrams. Specific forms of annotation are defined in subsequent paragraphs. The set of real-valued variables associated with a semantic diagram are those that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

or in any of the defining diagrams for abstract discrete states that appear in that diagram. Real-valued time-varying variables will be named using an italicized front. The initial values for the real-valued time-varying variables of a hybrid semantic diagram are undefined whenever they are not explicitly defined in narrative rules.

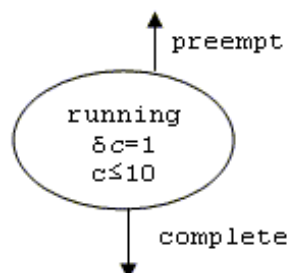
- (11) In addition to standard rational literals and arithmetic operators, expressions may also contain functions of discrete variables. The names of functions and discrete variables will begin with upper case letters. The semantics for function symbols and discrete variables will be defined using narrative rules. For example, the subexpression `Max(Compute_Execution_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_Execution_Time`.
- (12) Edges may be annotated with assignments of values to variables associated with the semantic diagram. When a transition occurs over an edge, the values of the variables are set to the assigned values. For example, in the following diagram, the values of the variables `c` and `t` are set to 0 when the component transitions into the `ready` discrete state.



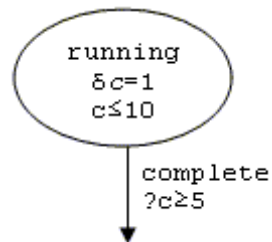
- (13) Discrete states may be annotated with expressions that define the possible rates of change for real-valued variables during the duration of time a component is in that discrete state. The rate of a variable is denoted using the symbol δ , for example $\delta x=[0,1]$ (the rate of the variable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



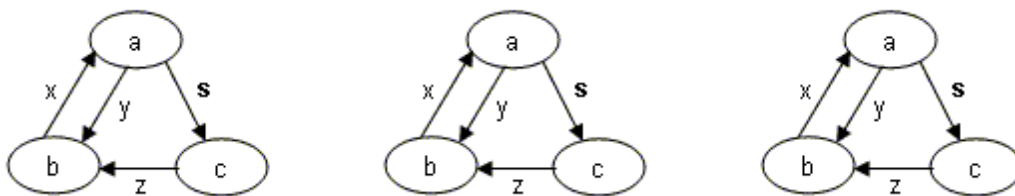
- (14) A discrete state may be annotated with Boolean-valued expressions called invariants of that discrete state. In this standard, all semantic diagrams are defined so that the values of the variables will always satisfy the invariants of a discrete state for every possible transition into that discrete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



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



- (16) A sequential semantic automaton defines semantics for a single component. A system may contain multiple components. The semantics of a system are defined to be the concurrent composition of the sequential semantic automata for each component. Except as described below, every component is represented by a copy of its defined semantic automaton. All discrete states and labels, all edges and labels, and all variables, are local to a component. The set of discrete states of the system is the cross-product of the sets of discrete states for each of its cross product components. The set of transitions that may occur for a system at any point in time is the union of the transitions that may occur at that instant for any of its components.
- (17) If an edge label appears in boldface, then a transition may occur over that edge only when a transition occurs over all edges having that same boldface label within the synchronization scope for that label. The synchronization scope for a boldface label is indicated in parentheses. For example, if a transition occurs over an edge having a boldface label with a synchronization scope of process, then every thread contained in that process in which that boldface label appears anywhere in its hybrid semantic diagram must transition over some edge having that label. That is, transitions over edges with boldface labels occur synchronously with all similarly labeled edge transitions in all components associated with the component with the specified synchronization scope as described in the narrative. Furthermore, every component in that synchronization scope that might participate in such a transition in any of its discrete states must be in one of those discrete states and participate in that transition. For example, when the synchronization scope for the edge label **s** is the same for all three of the following concurrent semantic automata, a transition over the edge labeled **s** may only occur when all three components are in their discrete states labeled *a*, and all three components simultaneously transition to their discrete states labeled *c*.



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

2 REFERENCES

2.1 Normative References

- (1) The following normative documents contain provisions that, through reference in this text, constitute provisions of this standard.
- (2) ISO/IEC/IEEE 24765:2017 Systems and Software Engineering – Vocabulary, Sep. 2017.
- (3) ISO/IEC 9945-1:2009, Information technology — Portable Operating System Interface (POSIX®) Base Specifications, Issue 7.
- (4) ISO/IEC 14519:2001 Information Technology – POSIX Ada Language Interfaces – Binding for System Application Program Interface (API).
- (5) ISO/IEC 8652:2012, Information Technology – Programming Languages – Ada.
- (6) ISO/IEC 9899:2018, Information Technology – Programming Languages – C.
- (7) Unified Modeling Language Specification, Dec 2017, version 2.5.1.
- (8) SAE AS5506C, Architecture Analysis & Design Language (AADL), January 2017.
- (9) SAE AS-5506/1A, Architecture Analysis & Design Language (AADL) Annex Volume 1, September 2015.

2.2 Informative References

- (1) The following informative references contain background information about the items with the citation.
- (2) [BNF 1960] NAUR, Peter (ed.), "Revised Report on the Algorithmic Language ALGOL 60," *Communications of the ACM*, Vol. 3 No. 5, pp. 299-314, May 1960.
- (3) [IFIP WG10.4-1992] IFIP WG10.4 on Dependable Computing and Fault Tolerance, 1992, J.-C. Laprie, editor, "Dependability: Basic Concepts and Terminology," Dependable Computing and Fault Tolerance, volume 5, Springer-Verlag, Wien, New York, 1992.
- (4) [Henz 96] "Theory of Hybrid Automata", Thomas A. Henzinger, Electrical Engineering and Computer Science, University of California at Berkley, *Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS)*, IEEE Computer Society Press, 1996, pp. 278-292

2.3 Terms and Definitions

- (1) Terms are introduced throughout this standard, indicated by *italic* type. Informational definitions of terms are given in Appendix B , Glossary. Definitions of terms used from other standards, such as ISO/IEC/IEEE 24765 Systems and Software Engineering – Vocabulary, Dec. 2010, ISO/IEC 9945-1:1996 [IEEE/ANSI Std 1003.1, 1996 Edition], *Information Technology – Portable Operating System Interface (POSIX)*, or IFIP WG10.4 *Dependability: Basic Concepts and Terminology* [IFIP WG10.4-1992], are so marked. Terms not defined in this standard are to be interpreted according to the Webster's Third New International Dictionary of the English Language. Terms explicitly defined in this standard are not to be presumed to refer implicitly to similar terms defined elsewhere. A full description of the syntax and semantics of the concept represented by the terms is found in the respective document sections, clauses, and subclauses.

3 ARCHITECTURE ANALYSIS & DESIGN LANGUAGE SUMMARY

- (1) This section provides an informative overview of AADL concepts, structure, and use. In this section the first appearance of a term that has a specific meaning in this standard will be italicized.
- (2) An AADL specification represents a component model of a computer system runtime architecture that consists of the *application software* (typically embedded, safety-critical, mission-critical, or performance-critical), and the *execution platform*, i.e., the computing hardware and the physical system. A *component* represents a part of a system and interacts with other components. A *system* is hierarchically composed of interacting components. An AADL specification consists of *package* declarations and *property set* declarations. AADL packages contain component specifications, i.e., AADL *component types* and *component implementations*, as well as *feature group types*, and *annex libraries*. This standard defines *data*, *subprogram*, *subprogram group*, *thread*, *thread group*, and *process* as application software component categories, *memory*, *bus*, *virtual bus*, *processor*, *virtual processor*, and *device* as execution platform component categories, and *system* and *abstract* as general compositional components. *Abstract* is a generic component category that can be refined into any of the other categories. They form the core of the AADL modeling vocabulary.
- (3) A *component type* specifies an external interface that other components can operate against in terms of *features*, i.e., interaction points with other components, *flow specifications* from component inputs to component outputs, *modes* as operational states, and *properties* to characterize a component. Implementations of the component are required to satisfy this specification.
- (4) A *feature* describes an interface of a component through which control and data may be provided to or required from other components. Features can be *ports* to support directional flow of control and data, *subprograms* to represent procedure calls, and shared *access* to data, subprograms, subprogram groups, and bus components. Features can be grouped together into *feature groups*.
- (5) A *component implementation* specifies a realization of the component in terms of *subcomponents*, *connections* between the features of those subcomponents, *flows* across a sequence of subcomponents, *modes*, and *properties*. AADL allows multiple component implementations to be associated with a component type to represent component variants.
- (6) A *subcomponent* declares a component instance that is contained in another component by naming its *component classifier*, i.e., a component type or component implementation. The component hierarchy of a system instance is determined by recursively instantiating the subcomponents of a top-level system.
- (7) Any namable model element, e.g., components, features, modes, connections, flows, and subprogram calls, can have properties. Properties are used to represent attributes and other characteristics, such as the period and deadline of threads. When properties are associated with declarations of component types, component implementations, features, subcomponents, connections, flows, and modes, they apply to all respective instances within a system instance. The AADL also supports the specification of instance specific values of any unit in the containment hierarchy of a system instance. AADL tools may record these values for use in the analysis of the system instance or for use in the construction of new system instances.
- (8) *Property sets* are used to define *properties*, *property constants*, and *property types*. This standard defines a set of predeclared properties and property types. Additional properties and property types to support new forms of system analysis can be introduced through property sets. For example, a predeclared property is used to specify the period of a thread. An example of a user-defined property in a property set is a security level property.
- (9) AADL packages provide a library-like structure for organizing component classifiers, and feature group types, and annex libraries into separate namespaces, similar to Java packages that are used to organize Java class declarations. Packages can have a nested naming hierarchy, but this hierarchy does not impose any restrictions on whether a package is accessible by other packages. A component classifier in a package is referenced by qualifying its name with the package name. Only those classifiers that have been placed in the *public* section of a package are accessible to other packages. Furthermore, the packages being named in the reference must be listed in the *with* clause of the referencing package, i.e., a *with* clause limits the use of other packages in a given package.

- (10) AADL supports the specification of partial models such as models of the application software only, the hardware only, specification of a top-level architecture in terms of its subsystems without their realizations, and specification of component templates. Component templates are incompletely specified component classifiers that may be parameterized by component classifier, feature, and feature group type *prototypes*. These component templates may be later refined through component classifier *extension* declarations. Component classifier extensions can complete the component specification by supplying an actual for a prototype and by completing partial declaration through *refinement*. Component extension declarations can also specify new component classifiers by adding features, subcomponents, connections, flows, and properties. This allows conceptual and reference architectures to be specified and to be refined into fully specified runtime architectures, and partial system specifications to evolve into fully specified and configured systems including the deployment of application software on the computing hardware
- (11) Application software components model *source text*, *virtual address spaces*, *concurrent tasks* and their interactions. *Source text* can be written in a programming language such as Ada, C, or Java, or domain-specific modeling languages such as Simulink, SDL, ESTEREL, LUSTRE, and UML, for which executable code may be generated. The source text modeled by a software component may represent a partial application program or model (e.g., they form one or more independent compilation units as defined by the applicable programming language standard). Rules and permissions governing the mapping between AADL specification and source text depend on the applicable programming or modeling language standard. Predeclared component properties identify the source text container and the mapping of AADL concepts to source text declarations and statements. These properties also specify memory and execution times requirements and other known characteristics of the component.
- (12) AADL *data* components represent static data in source text. These data components can be accessed by one or more threads and processes; they do so by indicating that they require access to the external data component. Concurrent access to data is managed by the appropriate concurrency control protocol as specified by a property. Realizations of such protocols can be documented through an appropriate annex declaration, for example, expressed in a Behavior Annex subclause (see Annex Document B).
- (13) Data types in the source text are modeled by the declarations: data component type and data component implementation. Thus, a data component classifier represents the data type of data components, ports, and subprogram parameters. The Data Modeling Annex (see Annex Document D) provides guidance on how to approach data modeling with AADL.
- (14) The *subprogram* component models source text that is executed sequentially. Subprograms are callable from within threads and subprograms. Subprograms may require access to data components and may contain data subcomponents to represent local variables. Subprogram groups represent source code libraries.
- (15) AADL *thread* components model concurrent tasks or active objects, i.e., concurrent logical threads. Each logical thread represents an execution sequence through source text (or more exactly, through binary images produced from the compilation, linking and loading of source text). A scheduler manages the execution of a thread. Logical threads may be executed by separate operating system (OS) threads, or they may be combined into a single operating system thread. The dynamic semantics for a thread are defined in this standard using hybrid automata. The threads can be in states such as suspended, ready, and running. State transitions occur as a result of dispatch requests, faults, and runtime service calls. They can also occur if time constraints are exceeded. Error detection and recovery semantics are specified. Dispatch semantics are given for standard dispatch protocols such as periodic, sporadic, aperiodic, timed, hybrid, and background threads. Additional dispatch protocols may be defined. Threads can contain subprogram and data components, and provide or require access to data components.
- (16) AADL *thread groups* support structural grouping of threads within a process. A thread group may contain data, thread, and thread group subcomponents. A thread group may require and provide access to data components.
- (17) AADL *process* components model space partitions in terms of virtual address spaces containing source text that forms complete binary image (e.g. executable, software components, or partitions) as defined in the applicable programming language standard. The binary image produced by compiling and linking this source text must execute properly when loaded into a unique virtual address space. As processes do not represent concurrent tasks, they must contain at least one thread. Processes can contain thread groups, threads, and data components, and can access or share data components.

- (18) Execution platform components represent computing hardware components that are capable of scheduling threads, of enforcing specified address space protection at runtime, of storing source text code and data and of performing communication for application system connections.
- (19) AADL *processor* components are an abstraction of hardware and software that is responsible for scheduling and executing threads. In other words, a processor may include functionality provided by operating systems. Alternatively, operating systems can be modeled like application components. Processors can contain memory and require access to buses. Processors can support different scheduling protocols. Threads are bound to processors for scheduling and execution.
- (20) AADL *virtual processors* represent virtual machines or hierarchical schedulers. Threads can be bound to them. Virtual processors can be used in two ways. Processors and virtual processors can be subdivided into virtual processors by declaring virtual processor subcomponents. Virtual processors can also be declared separately and explicitly bound to processors and virtual processors.
- (21) AADL *memory* components model randomly accessible physical storage such as RAM or ROM. Memories have properties such as the number and size of addressable storage locations. Binary images of source text are bound to memory. Memory can contain nested memory components. Memory components require access to buses.
- (22) AADL *bus* components model communication channels that can exchange control and data between processors, memories, and devices. A bus is typically hardware that supports specific communication protocols, possibly implemented through software. Processors, memories, and devices communicate by accessing a shared bus. Buses can be directly connected to other buses. Logical connections between threads that are bound to different processors transmit their information across buses that provide the physical connection between the processors. Buses can require access to other buses.
- (23) AADL *virtual bus* components represent virtual channels or communication protocols that perform transmission within processors or across buses. Virtual buses can be subcomponents of buses and virtual buses, or virtual buses can be declared separately and explicitly bound to buses and virtual buses.
- (24) AADL *device* components model physical entities in the external environment, e.g., a GPS system, or entities that interface with an external environment, e.g., sensors and actuators as interface between a physical plant and a control system. Devices may represent a physical entity of the modeled system or its (simulated) software equivalent. Examples of devices are timers, which exhibit simple behavior, or a camera or GPS, which exhibit complex behavior. Devices are logically connected to application software components and physically connected to processors via buses. They cannot store nor execute external application software source text themselves, but may include driver software executed on a connected processor. A device requires access to buses.
- (25) AADL *system* components model hierarchical compositions of software and execution platform components. A system may directly contain data, subprogram, subprogram groups, process, memory, processor, virtual processor, bus, virtual bus, device, system as well as abstract subcomponents. Thread and thread group subcomponents must be declared in processes and are indirectly part of a system that contains these processes. A system component may require and provide access to data and bus components. Execution platform component can be system components in their own right and be modeled using system implementations. For example, a system implementation can be associated with a device that models a camera. This system implementation describes the internal of the camera in terms of the CCD sensor a device, a DSP processor, a general purpose processor as well as a software that implements the image processing and download capability of the camera.
- (26) AADL *abstract* components can be used to represent components early in the design process, prior to decide their actual category. They can be refined into any other categories as the model is elaborated further, pending such refinement respects AADL rules.

- (27) 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.
- (28) This standard defines several categories of features: *data port*, *event port*, *event data port*, *feature group*, *subprogram parameter*, and *provided* and *required access* to data, subprograms, buses and virtual buses. Data ports represent connection points for transfer of state data such as sensor data. Event ports represent connection points for transfer of control through raised events that can trigger thread dispatch or mode transition. Event data ports represent connection points for transfer of events with data, i.e., messages that may be queued. Feature groups support grouping of ports and other features, such that they can be connected to other components through a single connection. Provided subprogram access features represent entrypoints to code sequences in source text that is associated with a data type or a thread that can be called locally or remotely. Subprogram parameters represent in and out parameters of a subprogram. Data component access represents provided and required access to shared data. Bus (resp. virtual bus) component access represents provided and required access to buses (resp. virtual bus) for processors, memory, and devices.
- (29) AADL *connections* specify interaction between components at runtime. A semantic connection is represented by a set of one or more connection declarations that follow the component hierarchy from the ultimate connection source to the ultimate connection destination. For example, in Figure 1 there is a connection declaration from a thread out port in Thread1 to a containing process out port in Process3. This connection is continued with a connection declaration within System1 from Process3's out port to Process4's in port. The connection declaration continues within Process4 to the thread in port contained in Thread2. Collectively, this sequence of connections defines a single semantic connection between Thread1 and Thread2. Threads, processes, systems, and ports are shown in graphical AADL notation.

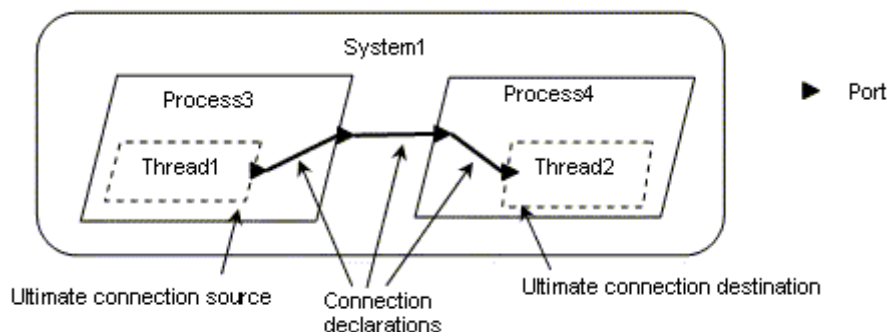


Figure 1 – Example Semantic Connections

- (30) *Flow specifications* describe externally observable flow of information in terms of application logic through a component. Such logical flows may be realized through ports and connections of different data types and a combination of data, event, and event data ports, as well as through data components. Flow specifications represent *flow sources*, i.e., flows originating from within a component, *flow sinks*, i.e., flows ending within a component, and *flow paths*, i.e., flows through a component from its incoming ports to its outgoing ports.
- (31) *Flow implementations* describe actual flow sequences representing flow specifications through components and sets of components across one or more connections. They are declared in component implementations. An *end-to-end flow* specifies a flow that starts within one subcomponent and ends within another subcomponent. Flow specifications, flow implementations, and end-to-end flows can have expected and actual values and flow-related properties, e.g., latency or rounding error accumulation.

- (32) An actual embedded system is represented by an instance of an AADL system implementation that consists of subcomponents representing the application software, the computing platform, and the physical environment.
- (33) An AADL specification may be used in a variety of ways by a variety of tools during a broad range of life-cycle activities, e.g., for documentation during preliminary specification, for schedulability or reliability analysis during design studies and during verification, for generation of system integration code during implementation. Note that application software components must be bound to execution platform components - ultimately threads to processors and binary images to memory in order for the system to be analyzable for runtime properties and the actual system to be constructed from the AADL specification. Many uses of an AADL specification need not be fully automated, e.g., some implementation steps may be performed by hand.
- (34) The AADL core language is extensible through property sets, *annex subclauses* and *annex libraries* that can be standardized or user-defined. Property sets extend AADL by introducing additional properties that can be associated with elements of a model, while annexes introduce sublanguages that can be used to annotate a model. *Annex subclauses* consist of annex-specific sublanguages whose constructs can be added to component types and component implementations. *Annex libraries* are declarations of reusable annex-specific sublanguage elements that are placed in AADL packages and can be referenced in annex subclauses. Examples of standardized extensions are the Error Modeling Annex (see Annex Document E) and the Behavior Modeling Annex (Annex Document B).

4 COMPONENTS, PACKAGES, AND ANNEXES

- (1) The AADL defines the following *categories* of components: *data*, *subprogram*, *subprogram group*, *thread*, *thread group*, *process*, *memory*, *bus*, *virtual bus*, *processor*, *virtual processor*, *device*, *system*, and *abstract*. The category *abstract* can be refined into any of the other categories. This section describes those aspects of components that are common to all AADL component categories. This section also describes packages as an organizing mechanism for component types and implementations. This section closes with the definition of annex subclauses and annex libraries.
- (2) A component represents some hardware or software entity that is part of a system being modeled in AADL. A component has a *component type*, which defines a functional interface. The component type acts as the specification of a component that other components can operate against. It consists of features, flows, and property associations.
- (3) A feature models a characteristic of a component that is visible to other components. Features are named, externally visible parts of the component type, and are used to exchange control and data via connections with other components. Features include ports to support directional flow of data and control, and subprograms including support for remote procedure call interactions. Features define parameters that represent the data values that can be passed into and out of subprograms. Features specify component access requirements for external data and bus components. Component access requirements for external subprograms support remote procedure call interactions.
- (4) A component has zero or more *component implementations*. Component implementations represent variants of a component that adhere to the same interface, but may have different property values. A component implementation specifies the realization of a component variant, i.e., an internal structure for a component as an assembly of subcomponents. Subcomponents are instantiations of *component classifiers*, i.e., component types and implementations.
- (5) Components are named and have properties. These properties have associated values that represent attributes and characteristics of a component.
- (6) Components can be declared in terms of other components by refining and extending existing component types and component implementations. This permits partially complete component type and implementation declarations to act as templates that may have explicit parameter (*prototype*) specifications. Such templates can represent a common basis for the evolution of a family of related component types and implementations.
- (7) This standard defines basic concepts and requirements for determining compliance between a component specification and a physical component. Within this framework, annexes to this standard will specify detailed compliance requirements for specific software programming, application modeling, and hardware description languages. This standard does not restrict the lower-level representation(s) used for software components, e.g., binary images, conventional programming languages, application modeling languages, nor does it restrict the lower-level representation(s) used for physical hardware component designs, e.g., circuit diagrams, hardware behavioral descriptions.

4.1 AADL Specifications

- (1) An AADL specification consists of package and property set declarations that are considered to be global declarations.
- (2) Packages provide a way for organizing collections of component type and implementation, feature group type, annex library declarations along with relevant property associations.
- (3) Property sets provide extensions to the core AADL in the form of new properties and property types in support of additional modeling and analysis capabilities. Similarly annex libraries and annex subclauses provide extension to the core AADL to annotate the base model through the use of sublanguages.

- (4) The content of packages, e.g., classifiers, can be referenced from anywhere by qualifying the classifier reference with the package name. The content of property sets, i.e., property type, property constant and property definitions, can be referenced by qualifying the property type, constant, or property reference with the property set name. References to predeclared properties do not require qualification by property set. Classifiers in other packages and properties in property sets can only be referenced with those packages and property sets are listed in the **with** declaration of a package or property set.
- (5) System instances are identified to processing tools and methodologies by referencing a system implementation component as the root of the system instance (see Section 13.1).

Syntax

```
AADL_specification ::=  
  ( package_spec | property_set ) +
```

Naming Rules

- (N1) An AADL specification has one *global* namespace. The package and property set identifiers reside in this space and must be unique.
- (N2) These package and property set identifiers qualify the names of individual elements contained in them when they are referenced.
- (N3) Package declarations represent labeled namespaces for component type, component implementation, feature group type, and annex library declarations.
- (N4) Property set declarations represent labeled namespaces for property type and property definition declarations.
- (N5) Packages and property sets may be separately stored. Those packages and property sets are considered to be part of the global namespace.
- (N6) Defining identifiers in AADL must not be one of the reserved words of the language (see Section 15.7).
- (N7) The AADL identifiers and reserved words can be in upper or lower case (or a mixture of the two) (see Section 15).
- (N8) The AADL does not require that an identifier be declared before it is referenced.

Semantics

- (6) An AADL specification provides a global namespace for packages and property sets. Items in packages and property sets can refer to items in other packages and property sets.
- (7) A package is used to introduce new component types, component implementations, feature group types, and annex libraries.
- (8) A property set is used to introduce new property types and properties (see section 11.1). They extend the predefined set of properties of the core AADL.
- (9) Declarations in an AADL specification can refer to packages and property sets declared in separately stored AADL specifications. This allows packages and property sets to be stored separately and used by multiple AADL specifications. Mechanisms for locating such separately declared packages and property sets are tool specific.

Processing Requirements and Permissions

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

4.2 Packages

- (1) A package provides a way to organize component types, component implementations, feature group types, and annex libraries into related sets of declarations by introducing separate namespaces. Package names are built using identifiers that are separated by double colons ("::"). This avoids the problem of duplicate names which might occur when packages are developed independently and then combined to model an integrated system. In other words, a::b1::c is distinct from a::b2::c. Packages cannot be declared inside other packages.
- (2) Packages have public and private sections. Classifier declarations in public sections are accessible to other packages, while classifiers in private sections can only be referenced within the private section of the same package. This allows classifiers used in the implementation of public components to be kept private.
- (3) Visibility to other packages is specified by a **with** declaration. Only classifiers in the packages listed in the **with** declaration can be referenced from within a given package. A **renames** declaration can be used to introduce a local alias to a qualified component type and feature group type reference.
- (4) Visibility to property sets is specified by a **with** declaration. Only properties, property types, and property constants defined in the property sets listed in the **with** declaration can be referenced from within a given package unless they are predeclared properties, property types, or property constants.

Syntax

```

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

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

package_name ::=
    { package_identifier :: }* package_identifier

none_statement ::= none ;

AADL_declaration ::=
    classifier_declaration
    | annex_library

classifier_declaration ::=
    component_classifier_declaration | feature_group_classifier_declaration

component_classifier_declaration ::=

```

```

component_type | component_type_extension |
component_implementation | component_implementation_extension

feature_group_classifier_declaration ::=
    feature_group_type | feature_group_type_extension

name_visibility_declaration ::=
    import_declaration |
    alias_declaration

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

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

```

Naming Rules

- (N1) A defining package name consists of a sequence of one or more package identifiers separated by a double colon ("::"). A defining package name must be unique in the global namespace.
- (N2) The **public** and **private** section of a package may be declared in separate package declarations; these two declarations introduce a single defining package name.
- (N3) Associated with every package is a *package namespace* that contains the names for all the elements defined within that package. This means that component types, feature group types, and defining entities declared in an annex library using an annex-specific sublanguage can be declared with the same name in different packages.
- (N4) The package namespace is divided into a public section and a private section. Items declared in the public section of the package namespace can be referenced from outside the package as well as within the package.
- (N5) The reference to an item declared in another package must be an item *name* qualified with a package name separated by a double colon ("::"). The package name must be listed in the `import_declaration` (**with**) of the package with the reference, i.e., in the `import_declaration` of the public section, if the reference is in the public section, and in the `import_declaration` of the public or private section if the reference is in the private section of the package. Only classifiers in the public package section can be referenced.

Note: References to items declared in the same package as the reference do not have to be qualified. If they are qualified the package is not required to be listed in the `import_declaration`.

- (N6) The reference to a property other than predeclared properties must be an property *name* qualified with a property set name separated by a double colon (":"). The property set name of a property reference in the properties section of package must be listed in the **with** declaration of the public section or in the private section if the package only contains a private section. Predeclared properties may be, but are not required to, be qualified by the property set name.
- (N7) The package name in a `import_declaration` must exist in the global name space.
- (N8) The property set identifier in a `import_declaration` must exist in the global name space.
- (N9) Items declared in the **private** section of the package can only be referenced from within the **private** section of the package.
- (N10) If the qualifying package identifier of a qualified reference is missing, the referenced component classifier, feature group type, or item in an annex library must exist in the same package as the reference.
- (N11) The package name referenced in an `alias_declaration` must exist in the global namespace and must be listed in the `import_declaration`.
- (N12) The classifier referenced by the `alias_declaration` must exist in the name space of the **public** section of the package being referenced by the `alias_declaration`.
- (N13) The classifier referenced by the `alias_declaration` must refer to a component type or a feature group type.
- (N14) The defining identifier of an `alias_declaration` must be unique in the namespace of the package containing the `alias_declaration`. If an `alias_declaration` defines an alias for a package then the alias name must not conflict with any package name listed in an `import_declaration` or that of the package containing the `alias_declaration`.
- (N15) The `alias_declaration` makes the publicly visible identifier of classifiers declared in another package accessible in the name space of the package containing the `alias_declaration`. If the alias declaration for a classifier does not include a defining identifier then the referenced classifier identifier is used as defining identifier and this identifier must be unique in the namespace of the package with the alias declaration.
- (N16) If the `alias_declaration` renames all publicly visible identifiers of component types and feature group types by naming the package and **all**, then all those identifiers must also be unique in the namespace of the package with the alias declaration.
- (N17) The identifiers introduced by the `alias_declaration` are only accessible within the package. When declared in the public package section, they can be referenced within the public and private package section, but not from other packages. When declared in the private package section, they can be referenced within the private package section only.
- (N18) The alias declared for a component type can be used instead of a qualified component type in a reference to a component implementation.

Legality Rules

- (L1) The defining package name following the reserved word **end** must be identical to the defining package name following the reserved word **package**.
- (L2) For each package there may be at most one **public** section declaration and one **private** section declaration. These two sections may be declared in a single package declaration or in two separate package declarations.
- (L3) A component implementation may be declared in both the public and private part of a package. In that case the declaration in the public part may only contain a **properties** subclause and a **modes** subclause.

- (L4) The component category in an alias declaration must match the category of the referenced component type.

Semantics

- (5) A package provides a way to organize component type declarations, component implementation declarations, feature group types, and annex libraries into related sets of declarations along with relevant property associations. It provides a namespace for component types, feature group types, and annex libraries with the package name acting as a qualifier. Nested package names allow for unique package naming conventions that address potential name conflicts in component type and implementation names when independently developed AADL specifications are combined. Note that component implementations are named relative to component types. Thus, qualified component type names act as unique qualifier for component implementation names. Packages can be organized hierarchically by giving them nested package names. These package names represent absolute paths from the root of the package hierarchy.
- (6) Packages have a **public** and a **private** section. Declarations in the **public** section are visible outside the package, i.e., names declared in the **public** part can be referenced by declarations in other packages. Declarations in the **private** section are visible only within **private** section of the package, i.e., they cannot be referenced from the **public** section or from other packages.
- (7) Component type and component implementation declarations model execution platform and application software components of a system. A component type denotes externally visible characteristics of a component, i.e., its features and its properties. A component implementation denotes the internal structure, operational modes, and properties of a component. A component type can have several component implementations. This can be used for example to model product line architectures running on different execution platforms. Packages allow such declarations to be organized into separate namespaces.
- (8) Feature group types provide the definition of an interface to a component that represents a collection of features or feature groups defined within the component implementation (see Section 5.3). This group of features may be accessed externally as a single unit.
- (9) A component implementation can be declared in both the public and private section of a package. If it is declared in both, then the public declaration is limited to containing property associations and modes and only those items are visible outside the package. This allows component implementations to be made visible to other packages as variants of the same component type, while the details of the implementation, i.e., its realization expressed by the subcomponents and connections subclauses, are hidden in the private part. The two declarations represent the same component implementation.
- (10) When a component implementation is declared as an extension of another component implementation and it is declared in both the public and private section of a package, then the **extends** is specified with the public section and the extension may include prototype bindings. The component implementation in the private section is considered to complete the declaration in the public section, i.e., its name can be interpreted as reference to the defining name of the component implementation declaration in the public portion.
- (11) An `import_declaration` specifies which packages and property sets can be named in qualified references to items in other packages or property sets. Packages can initially be declared with `import_declaration` without classifiers to set up an initial collection of package with use restrictions on other packages.
- (12) An `alias_declaration` introduces local identifiers as short names for long names. It does so for package names and for classifier type references qualified by a package name. The short name may differ from the identifier of the long name to avoid name conflicts.
- (13) Property associations declared in the properties section of a package are associated with the package represented by the declaration. Packages with separate public and private package declarations can have different property values for the same property.

Examples

```
package Aircraft::Cockpit
public
```

```

with Avionics::DataTypes, Safety_Properties;
AirData renames data Avionics::DataTypes::AirData;
system MFD
features
    Airdata: in data port AirData;
properties
    Safety_Properties::Safety_Criticality => high;
end MFD;
end Aircraft::Cockpit;

```

4.3 Component Types

- (1) A component type specifies the external interface of a component that its implementations satisfy. It contains declarations that represent features of a component and property associations. Features of a component are ports, feature groups, required access to externally provided data, subprogram, and bus components, and parameter declarations for the specification of the data values that flow into and out of subprograms. The ports and feature groups of a component can be connected to compatible ports or subprograms of other components through connections to represent control and data interaction between those components. Required access to an external subcomponent, such as data, subprogram, or bus, is connected via required and provided access to a subcomponent of the specified component classifier.
- (2) Component types can declare flow specifications, i.e., logical flows of information from its incoming ports to its outgoing ports that are realized by their implementations.
- (3) Component types can declare modes and mode transitions of a component. These modes and mode transitions are common to all implementations of the component. This allows for mode-specific property values to be associated with the component type, its features, and its flows.
- (4) Component types can be declared in terms of other component types, i.e., a component type can *extend* another component type – inheriting its declarations and property associations. If a component type extends another component type, then features, flows, and property associations can be added to those already inherited. A component type extending another component type can also refine the declaration of inherited feature and flow declarations by more completely specifying partially declared component classifiers of features and by associating new values with properties. A component type cannot be an extension of multiple component types, i.e., multiple inheritance is not supported for AADL components
- (5) Component types can declare *prototypes*, i.e., classifier parameters that are used in features. The prototype bindings are supplied when the component types is being extended or used in subcomponent declarations.
- (6) Component type extensions form an *extension hierarchy*, i.e., a component type that extends another component type can also be extended. We use AADL graphical notation (see Appendix D) to illustrate the extension hierarchy in Figure 2. For example, component type GPS extends component type Position System inheriting ports declared in Position System. It may add a port, refine the data type classifier of a port incompletely declared in Position System, and overwrite the value of one or more properties. Component types being extended are referred to as *ancestors*, while component types extending a component type are referred to as *descendents*.

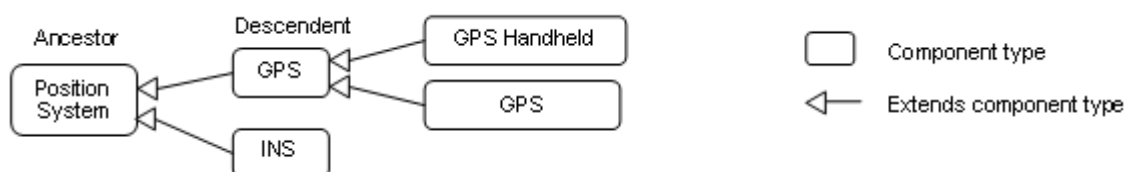


Figure 2 – Component Type Extension Hierarchy

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

Syntax

```

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

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

component_category ::=
    abstract_component_category
    | software_category
    | execution_platform_category
    | composite_category

abstract_component_category ::= abstract

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

```

```

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

composite_category ::= system

unique_component_type_reference ::=
    [ package_name :: ] component_type_identifier

```

NOTE: 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.

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

Naming Rules

- (N1) The defining identifier for a component type must be unique in the namespace of the package within which it is declared.
- (N2) Each component type has a local namespace for defining identifiers of prototypes, features, modes, mode transitions, and flow specifications. That is, defining prototype, defining feature, defining modes and mode transitions, and defining flow specification identifiers must be unique in the component type namespace.
- (N3) The component type identifier of the ancestor in a component type extension, i.e., that appears after the reserved word **extends**, must be defined in the specified package namespace. If no package name is specified, then the identifier must be defined in the namespace of the package the extension is declared in.
- (N4) When a component type extends another component type, a component type namespace includes all the identifiers in the namespaces of its ancestors.
- (N5) A component type that extends another component type does not include the identifiers of the implementations of its ancestors.
- (N6) The defining identifier of a feature, flow specification, mode, mode transition, or prototype must be unique in the namespace of the component type.
- (N7) The refinement identifier of a feature, flow specification, or prototype refinement refers to the closest refinement or the defining declaration of the feature going up the component type ancestor hierarchy.
- (N8) The prototypes referenced by prototype binding declarations must exist in the local namespace of the component type being extended.
- (N9) Mode transitions declared in the component type may not refer to event or event data ports of subcomponents.

Legality Rules

- (L1) The defining identifier following the reserved word **end** must be identical to the defining identifier that appears after the component category reserved word.
- (L2) The **prototypes**, **features**, **flows**, **modes**, and **properties** subclauses are optional. If a subclause is present but empty, then the reserved word **none** followed by a semi-colon must be present.
- (L3) The category of the component type being extended must match the category of the extending component type, i.e., they must be identical or the category being extended must be **abstract**.

- (L4) The classifier being extended in a component type extension may include prototype bindings. There must be at most one prototype binding for each prototype, i.e., once bound a prototype binding cannot be overwritten by a new binding in a component type extension.
- (L5) A component type must not contain both a `requires_modes_subclause` and a `modes_subclause`.
- (L6) If the extended component type and an ancestor component type in the extends hierarchy contain modes subclauses, they must both be `requires_modes_subclause` or `modes_subclause`.

Standard Properties

Classifier Substitution Rule: **inherit enumeration** (Classifier Match, Type Extension, Signature_Match)

Prototype Substitution Rule: **inherit enumeration** (Classifier Match, Type Extension, Signature Match)

Semantics

- (9) A component type represents the interface specification of a component, i.e., the component category, prototypes, features, flow specifications, modes, mode transitions, and property values of a component. A component implementation of this component type denotes a component, existing or potential, that is compliant with the component type declaration. Component implementations are expected to satisfy these externally visible characteristics of a component. The component type provides a contract for the component interface that users of the component can depend on.
- (10) The component categories are: data, subprogram, subprogram group, thread, thread group, and process (software categories); processor, virtual processor, bus, virtual bus, memory, and device (execution platform categories); system (compositional category), and abstract component (compositional category). The semantics of each category will be described in sections 4 to 7.
- (11) Features of a component are interaction points with other components, i.e., ports and feature groups; subprogram parameters; data component access, subprogram access, and bus access. Ports represent directional flow of data and events between components, feature groups represent groups of features that are connected to another component, data component access represents access to shared data components, subprogram access represents access to a subprogram by a caller, and bus access represents access to a bus from processor, memory, device, and other bus components to establish hardware connectivity. Features are further described in Section 8.
- (12) 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.
- (13) Mode declarations define modes of the component that are common to all implementations. As a result, component types can have mode-specific property values. Other components can initiate mode transitions by supplying events to incoming event ports of a component. Mode transitions specify which event ports affect their transition. A component type extension can add modes and associate properties to existing modes.
- (14) A `requires_modes_subclause` specifies a set of modes that the component expects to inherit from its containing component. In this case, the component can utilize these modes for mode-specific property values and for **in modes** declarations of subcomponents and connection, but mode transition behavior is determined by the containing component, whose modes are made accessible to the component.
- (15) 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.
- (16) A component type may also be declared with *prototypes*, indicating that the component type is incomplete and can be parameterized. Classifiers and features can be supplied through prototype bindings to complete such a

component type template as part of a component type extension, component implementation extension, or when used as a component type reference, e.g., in feature declarations or subcomponent declarations. Once bound a prototype cannot be rebound. The use of incomplete declarations is particularly useful during early design stages where details may not be known or decided.

- (17) A component type can be declared as an extension of another component type resulting in a component type extension hierarchy, as illustrated in Figure 2. A component type extension can refine an abstract component type to one of the concrete component categories. A component type extension inherits the features, flow specifications, modes, mode transitions, prototypes, and properties of the component type being extended. For annex subclauses, each annex defines whether annex declarations are inherited. A component type extension can contain refinement declarations to permit incomplete feature declarations to be completed and new property values to be associated with features and flow specification declared in a component type being extended. In addition, a type extension can add feature declarations, flow specifications, modes, mode transitions, and property associations.
- (18) A component type being extended may include prototype binding declarations. If prototype bindings are declared for a subset of the prototypes, then only the prototypes without binding can be bound when referencing the component type extension. This supports evolutionary development and modeling of system families by declaring partially complete component types that get refined in extensions.
- (19) Properties are predefined for each of the component categories and will be described in the appropriate sections. See Section 11.3 regarding rules for determining property values.

Examples

```
package TypeExample
public
system File_System
features
    -- access to a data component
    root: requires data access FileSystem::Directory.hashcd;
end File_System;

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

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

```

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

```

4.4 Component Implementations

- (1) A *component implementation* represents the realization of a component in terms of subcomponents, their connections, flow sequences, properties, component modes and mode transitions. Flow sequences represent implementations of flow specifications in the component type, or end-to-end flows to be analyzed. Modes represent alternative operational modes that may manifest themselves as alternate configurations of subcomponents, connections, call sequences, flow sequences, and property values.
- (2) A component type can have zero, one, or multiple component implementations. If a component type has zero component implementations, then it is considered to be a leaf in the system component hierarchy. For example, a partial AADL model may have processes as components without realization, while a task level AADL model expand to threads as leaves. If no implementation is associated then the properties on the component types provide information about the component for analysis and system generation.
- (3) A component implementation can be declared as an extension of another component implementation. In that case, the component implementation inherits the declarations of its ancestors as well as its component type. A component implementation extension can refine inherited declarations, and add subcomponents, connections, subprogram call sequences, flow sequences, mode declarations, and property associations.
- (4) Component implementations can declare *prototypes*, i.e., classifier parameters that are used in subcomponent declarations. The prototype bindings are supplied when the component implementation is being extended or used in subcomponent declarations.
- (5) Component implementations build on the component type *extension hierarchy* in two ways. First, a component implementation is a realization of a component type (shown as dashed arrows in Figure 3). As such it inherits features and property associations of its component type and any component type ancestor. Second, a component implementation declared as extension inherits subcomponents, connections, subprogram call sequences, flow sequences, modes, property associations, and annex subclauses from the component implementation being extended (shown as solid arrows in Figure 3). A component implementation can extend a component implementation that in turn extends another component implementation, e.g., in Figure 3 *GPS.Handheld* extends *GPS.Basic* and is extended by *GPS_Secure.Handheld*. Component implementations higher in the extension hierarchy are called ancestors and those lower in the hierarchy are called descendants. 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.

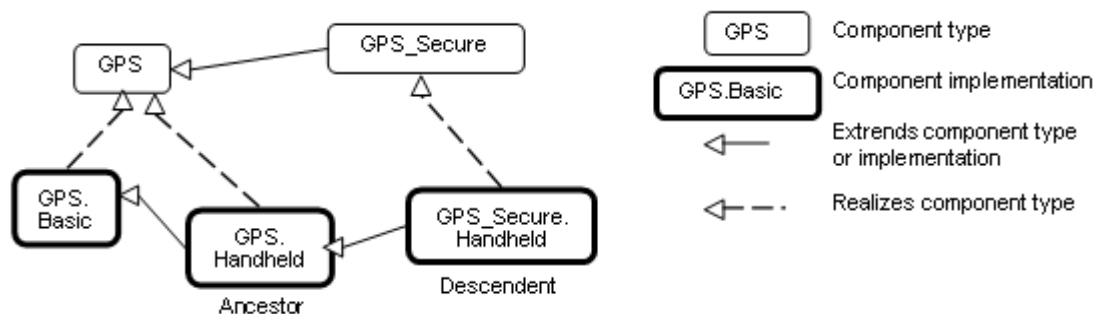


Figure 3 – Extension Hierarchy of Component Types and Implementations

Syntax

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

```
component_implementation_name ::=
    component_type_identifier . component_implementation_identifier
```

```
component_implementation_extension ::=
    component_category implementation
    defining_component_implementation_name
extends unique_component_implementation_reference [ prototype_bindings ]
[ prototypes ( { prototype | prototype_refinement }+ | none_statement ) ]
[ subcomponents
    ( { subcomponent | subcomponent_refinement }+ | none_statement ) ]
[ internal features { internal_feature }+ ]
[ processor features { processor_feature }+ ]
[ calls ( { subprogram_call_sequence }+ | none_statement ) ]
[ connections
    ( { connection | connection_refinement }+ | none_statement ) ]
[ flows ( { flow_implementation |
            end_to_end_flow | end_to_end_flow_refinement }+
            | none_statement ) ]
[ modes_subclause ]
[ properties ( { property_association | contained_property_association }+
                | none_statement ) ]
{ annex_subclause }*
```

```
end    defining_component_implementation_name ;
```

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

NOTE: 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.

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

Naming Rules

- (N1) A component implementation name consists of a component type identifier and a component implementation identifier separated by a dot ("."). The first identifier of the defining component implementation name must name a component type that is declared in the same package as the component implementation, or name an *alias* to a component type in another package.
- (N2) The defining identifier of the component implementation must be unique within the local namespace of the component type.
- (N3) Every component implementation defines a *local namespace* for all defining identifiers of prototypes, subcomponents, subprogram calls, connections, flows, and modes declared within the component implementation. The defining identifier of a prototype, subcomponent, subprogram call, connection, flow, or mode must be unique within this namespace. For example, a subcomponent and a mode cannot have the same defining identifier within the same component implementation, or a mode with the same defining identifier cannot be declared in a component type and a component implementation.
- (N4) This local namespace inherits the namespace of the associated component type, i.e., defining identifiers must be unique within the local namespace and also within the component type namespace.
- (N5) Refinement identifiers of prototype, subcomponent, and connection refinements must exist in the local namespace of an ancestor component implementation.
- (N6) In a component implementation extension, the component type identifier of the component implementation being extended, which appears after the reserved word **extends**, must be the same as or an ancestor of the component type of the extension. The component implementation being extended may exist in another package. In this case the component implementation name is qualified with the package name.
- (N7) When a component implementation **extends** another component implementation, the local namespace of the extension is a superset of the local namespace of the ancestor. That is, the local namespace of a component implementation inherits all the identifiers in the local namespaces of its ancestors (including the identifiers of their respective component type namespaces).
- (N8) Within the scope of the component implementation, subcomponent declarations, connections, subprogram call sequences, mode transitions, and property associations can refer directly to identifiers in the local namespace, i.e., to declared prototypes, subcomponents, connections, and modes, as well as to required bus, data, subprogram, and subprogram group subcomponents and features declared in the associated component type.
- (N9) The prototype referenced by the prototype binding declaration must exist in the local namespace of the component implementation being extended. In other words, prototype binding declarations may bind prototypes of the component type and of the component implementation.

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

renames LM::GPS allows the type GPS to be named without a qualifying package name. This allows a system implementation to be added to GPS in the package LM1.

package LM1 **public**

renames LM::GPS;

system implementation GPS.newimpl

Legality Rules

- (L1) The pair of identifiers separated by a dot (".") following the reserved word **end** must be identical to the pair of identifiers following the reserved word **implementation**.
- (L2) The **prototypes**, **subcomponents**, **connections**, **calls**, **flows**, **modes**, and **properties** subclauses are optional. If they are present and the set of feature or required subcomponent declarations or property associations is empty, **none** followed by a semi-colon must be present in that subclause.
- (L3) The category of the component implementation must be identical to the category of the component type for which the component implementation is declared.
- (L4) If the component implementation extends another component implementation, the category of both must match, i.e., they must be identical or the category being extended must be **abstract**.
- (L5) The classifier being extended in a component implementation extension may include prototype bindings. There must be at most one prototype binding for each unbound prototype.
- (L6) If the component type of the component implementation contains a `requires_modes_subclause` then the component implementation must not contain any modes subclause.
- (L7) If modes are declared in the component type, then modes cannot be declared in component implementations.
- (L8) If modes or mode transitions are declared in the component type, then mode transitions can be added in the component implementation. These mode transitions may refer to event or event data ports of the component type and of subcomponents.
- (L9) The category of a subcomponent being refined must match the category of the refining subcomponent declaration, i.e., they must be identical or the category being refined must be **abstract**.
- (L10) For all other refinement declarations the categories must match (see the respective sections).
- (L11) Component implementations and component implementation extensions must not refine prototypes declared in a component type.

Standard Properties

Classifier Substitution Rule: **inherit enumeration** (Classifier Match, Type Extension, Signature Match)

Prototype Substitution Rule: **inherit enumeration** (Classifier Match, Type Extension, Signature Match)

Classifier Matching Rule: **inherit enumeration** (Classifier_Match, Equivalence, Subset, Conversion, Complement)

Semantics

- (7) A component implementation represents the internal structure of a component through subcomponent declarations. Interaction between subcomponents is expressed by the connections, flows, and subprogram call sequences. Mode declarations represent alternative runtime configurations (internal structure) and alternative execution

behavior (interaction between subcomponents). A component implementation also has property values to express its non-functional attributes such as safety level or execution time which can also vary by mode.

- (8) Each component implementation is associated with a component type and provides a realization of its features (interface). A component type can have multiple implementations. A component implementation can be viewed as a component variant with the same interface but with differing property values that characterize the differences between implementations without exposing the internals of their realization. This can be achieved by placing the component implementation declaration without subcomponents and connections in the public section of an AADL package, and a component implementation declaration with subcomponents and connections for the same component in the private section of an AADL package.
- (9) The actual system being modeled by component types and component implementations may contain subcomponents, some of which may contain subcomponents themselves. The subcomponent containment hierarchy reflects the actual system structure.
- (10) A component implementation that is an extension of another inherits all prototypes, subcomponents, connections, subprogram call sequences, flow sequences (flow implementations and end-to-end flows), modes, mode transitions, and property associations from its ancestors as well as features, and associations, from its associated component type (and that component type's ancestors). For annex subclauses each annex defines whether annex declarations are inherited.
- (11) A component implementation extension can refine prototypes and subcomponents previously declared in ancestor component implementations by supplying component classifiers, and by associating new property values. A component implementation extension can refine connections, flows, and modes of its ancestor component implementations by associating new property values. A component implementation extension can refine features of its associated component type (and that component type's ancestors) by associating new property values to them. A component implementation being extended may include declaration of prototype bindings. If prototype bindings are declared for a subset of the prototypes, then only the prototypes without binding can be bound in component implementation extensions.

- (12) A component implementation extension can also add prototypes, subcomponents, connections, subprogram call sequences, flow sequences, modes, property associations, and annex subclauses. This extension capability supports evolutionary development and modeling of system families by declaring partially complete component implementations that get refined in extensions.
- (13) A descendent component implementation is said to contain all subcomponents whose identifiers appear in its local namespace, i.e., subcomponents declared in the component implementation and any of its ancestors. In other words, an instance of a component implementation extension contains instances of declared and inherited subcomponents, features, connections, subprogram call sequences, flow sequences, and modes.
- (14) Properties are predefined for each of the component categories and will be described in the appropriate sections. See Section 11.3 regarding rules for determining property values.

Examples

```
package ImplementationExample
public
data Bool_Type end Bool_Type;

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

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

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

4.5 Subcomponents

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

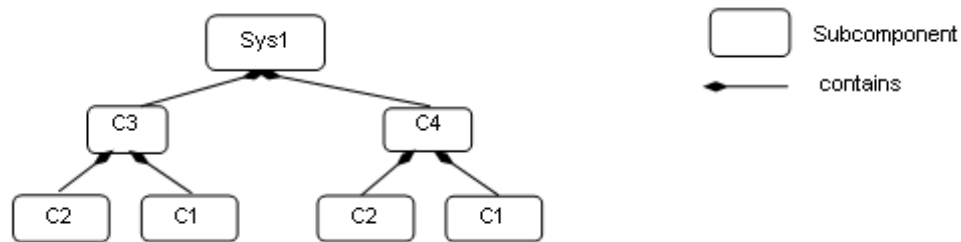


Figure 4 – Component Containment Hierarchy

- (2) A subcomponent declaration may resolve required subcomponent access declared in the component type of the subcomponent. For details on required subcomponent access see Section 8.4 and following.
- (3) A subcomponent can be declared to apply to specific modes (rather than all modes) defined within the component implementation.
- (4) Subcomponents can be refined as part of component implementation extensions. Refinement allows classifier references to be completed, abstract subcomponents to be refined into one of the concrete categories, and subcomponent property values to be associated. The resulting refined subcomponents can be refined themselves.
- (5) An array of subcomponents can be declared to represent a set of subcomponents with the same component type. This array may have one or more dimensions.

Syntax

subcomponent ::=

```

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

subcomponent_refinement ::=

```

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



```

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

```

```

unique_component_classifier_reference ::=
  ( unique_component_type_reference
    | unique_component_implementation_reference )

```

```

array_dimensions ::=
  { array_dimension }+

```

```

array_dimension ::=
  [ [ array_dimension_size ] ]

```

```

array_dimension_size ::=
  numeral | unique_property_constant_identifier

```

```

array_element_implementation_list ::=
  ( unique_component_implementation_reference [ prototype_bindings ]
    { , unique_component_implementation_reference [ prototype_bindings ] }* )

```

-- array selection used in contained property association and references

```

array_selection_identifier ::=
  identifier array_selection

```

```

array_selection ::=
  { [ selection_range ] }+

```

```

selection_range ::=
  numeral [ .. numeral ]

```

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

Naming Rules

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

- (N2) The defining identifier of a subcomponent refinement must exist as a defining subcomponent identifier in the local namespace of an ancestor component implementation.
- (N3) The component type identifier or the component implementation name of a component classifier reference must exist in the package namespace.
- (N4) The prototype identifier of a prototype reference must exist in the local name space of the component implementation.
- (N5) The prototype referenced by the prototype binding declarations must exist in the local namespace of the component classifier being referenced.
- (N6) The modes named in the **in modes** statement of a subcomponent must refer to modes in the component implementation that contains the subcomponent or its component type. The modes named in the **in modes** statement of a property association of a subcomponent must refer to modes of the subcomponent, or in the case of a contained property association to modes of the last component in the component path (see Section 11.3).

Legality Rules

- (L1) The category of the subcomponent declaration must match the category of its corresponding component classifier reference or its prototype reference, i.e., they must be identical, or in the case of a classifier reference the referenced classifier category may be *abstract*.
- (L2) The component classifier reference of a subcomponent declaration may include prototype bindings for a subset or all of the component classifier prototypes. This represents an unnamed component classifier extension of the referenced classifier.
- (L3) In a subcomponent refinement declaration the component category may be refined from **abstract** to one of the concrete component categories. Otherwise the category must be the same as that of the subcomponent being refined.
- (L4) The `Classifier_Substitution_Rule` property specifies the rule to be applied when a refinement supplies a classifier and the original subcomponent declaration already has a component classifier. This property can be applied to individual subcomponents or features, or it can be inherited from classifiers. The following rules are supported:
 - `Classifier_Match`: The component type of the refinement must be identical to the component type of the classifier being refined. If the original declaration specifies a component implementation, then any implementation of that type can replace this original implementation. This is the default rule.
 - `Type_Extension`: Any component classifier whose component type is an extension of the component type of the classifier in the subcomponent being refined is an acceptable substitute.
 - `Signature_Match`: The component type of the refinement must match the signature of the component type of the classifier being refined.
- (L5) In the case of a signature match, the component type of the subcomponent being refined must have a subset of the features of the component type in the refinement. The features are compared by name matching; the feature categories and direction (in data port, provides data access, etc.) must be the same and any feature classifier must match according to rules defined for `Classifier_Match`. In addition, if flow specifications are present in the component type being refined, then the component type of the refinement must have at least the same set of flow specifications. Flow specifications with the same name must have the same source and destination ports.
- (L6) The component category and optional component classifier or prototype reference can be followed by a set of array dimensions to define the subcomponent as an array of actual subcomponents.
- (L7) The array size specification for the dimensions is optional. In this case the array declaration is considered incomplete. If the size of the array dimension is specified it must be specified for all dimensions in the same declaration.
- (L8) When refining a subcomponent array the number of dimensions of the array cannot be changed, but the array size can be specified for each dimension if it was not specified in the subcomponent declaration being refined.

- (L9) When the subcomponent is declared as an array with array dimension sizes then a list of component implementations can be supplied, one for each element of the array. Different implementations of the same component type can be chosen. The number of elements in the list must correspond to the number of elements in the component array. In the case of multi-dimensional arrays, the list elements are assigned by incrementing the index of the last dimension first.
- (L10) Selecting index ranges in one or more dimensions of an array is only possible if the size of the array for these dimensions is already defined. The index range of a dimension is from 1 to the size of the dimension. Specification of array index ranges is limited to the **applies to** subclause of contained property associations. Specification of a single array element is limited to the **applies to** subclause of contained property associations and to the values of **reference** properties.
- (L11) An array element implementation list is valid only if (a) the subcomponent classifier is a component type and (b) all component implementations in the list are implementations of the specified type.

Consistency Rules

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

Standard Properties

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

Acceptable_Array_Size: **list of** Size_Range

Semantics

- (6) Subcomponents declared in a component implementation are considered to be contained in the component implementation. Contained subcomponents are instantiated when the containing component implementation is instantiated. Thus, the component containment hierarchy describes the hierarchical structure of the actual system.
- (7) A component implementation can contain *incomplete* subcomponent declarations, i.e., subcomponent declarations with no component classifier references, or if the component classifier reference only consists of a component type name for a component type with more than one component implementation. A subcomponent declaration is also incomplete when it consists of the declaration of an array of subcomponents for which the array sizes are not specified. This is particularly useful during early design stages where details may not be known or decided. Such incomplete subcomponent declarations can be refined in component implementation extensions.
- (8) A subcomponent declaration can be parameterized by referring to a prototype. In this case the component category and component classifier bound to the prototype is used when the system is instantiated.
- (9) A subcomponent declaration can reference a component classifier with prototype bindings. The prototype binding can refer to other classifiers or to a prototype of the component type or implementation that contains the subcomponent. In the latter case, the prototype actual is passed down levels of the component hierarchy and effectively allows the system subcomponents to be configured from a higher level component.
- (10) A component classifier reference with prototype bindings that refer to component classifiers effectively is an unnamed extension of the classifier being referenced. In other words, it could have been declared as a component type or component implementation extension with a new defining identifier and this identifier could have been referenced in the subcomponent declaration. Two unnamed component classifier extensions are not considered to be extensions of each other.
- (11) The optional `component_in_modes` subclause specifies the modes in which the subcomponent is active. An `component_in_modes` in a subcomponent refinement replaces previously specified subsets of modes. A subcomponent or subcomponent refinement without `component_in_modes` specifies that the subcomponent is active in all modes. The `component_in_modes` refer to modes of the component implementation that contains the subcomponent or to the modes of its component type. The `component_in_modes` may map mode identifiers

of the containing component to the mode identifiers specified in the **requires modes** clause of the subcomponent's component type (see Section 12).

- (12) A subcomponent can have property values associated to itself, or a contained property association can be declared for one of the subcomponents in its containment hierarchy, as well as those subcomponents' features, modes, subprogram call sequences, connections, and flows, or model elements in any annex subclause of a subcomponent (see Section 11.3). Subcomponent refinements may declare property associations – that override the property values declared in the subcomponent being refined. Property associations can have `in_modes` statements that refer to modes of the component implementation that contains the subcomponent, or in the case of contained property associations also to modes of the last subcomponent named in the path of the **applies to** (see Section 11.3).
- (13) The arrays of subcomponents are used to simplify the declaration of a multiplicity of subcomponents with the same classifier without declaring each of them separately. If a size of a subcomponent array is not known the array is incomplete and is assumed to have one element for the purpose of system instances of incomplete models. A subcomponent array can only be refined by adding array sizes to the dimensions if they are without a size.
- (14) All elements of a subcomponent array have the same component classifier, i.e., they are of the same kind. A subcomponent array can also be declared to have the same component type, but its elements vary in their implementation, e.g., to represent variants in an N-Version redundancy pattern.
- (15) A property association declared with a subcomponent array applies to each element in the array. Contained property associations declared in the enclosing component implementation can be used to associate different property values to different elements or subsets of the subcomponent array.

Processing Requirements and Permissions

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

Examples

- (17) The example illustrates modeling of source text data types as data component types without any implementation details. It illustrates the use of **package** to group data component type declarations. It illustrates both component classifier references to component types and to component implementations. It illustrates the use of ports as well as required and provided data access, and required subprogram access. In that context it illustrates the ways of resolving required access.

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

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

  data implementation Sample_Set.impl
```

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

data Dynamic_Sample_Set extends Sample_Set
end Dynamic_Sample_Set;

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

package SamplingTasks
public
with Sampling;
thread Init_Samples
features
    OrigSet : requires data access Sampling::Sample_Set;
    SampleSet : requires data access Sampling::Sample_Set;
end Init_Samples;

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

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

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

```
process Sample_Manager
```

```
features
```

```
    Input_Sample: in event data port Sampling::Sample;
    External_Samples: requires data access Sampling::Sample_Set;
    Result_Sample: out event data port Sampling::Sample;
```

```
end Sample_Manager;
```

```
process implementation Sample_Manager.Slow_Update
```

```
subcomponents
```

```
    Samples: data Sampling::Sample_Set;
    Init_Samples : thread Init_Samples;
    -- the required access is resolved to a subcomponent declaration
    Collect_Samples: thread Collect_Samples.Batch_Update;
    Distribute: thread Distribute_Samples;
    Sample_Filter: subprogram Sample_Subprogram.Simple;
```

```
connections
```

```
    ISSConn: data access Samples <-> Init_Samples.SampleSet;
    ISOSConn: data access External_Samples <-> Init_Samples.OrigSet;
    CSSConn: data access Samples <-> Collect_Samples.SampleSet;
    CSISConn: port Input_Sample -> Collect_Samples.Input_Sample;
    DSSConn: data access Samples <-> Distribute.SampleSet;
    DUSConn: port Distribute.UpdatedSamples -> Result_Sample;
    CSFRConn: subprogram access Sample_Filter <-> Collect_Samples.Filtering_Routine;
```

```
end Sample_Manager.Slow_Update;
```

```
subprogram Sample_Subprogram
```

```
end Sample_Subprogram;
```

```
subprogram implementation Sample_Subprogram.Simple
```

```
end Sample_Subprogram.Simple;
```

```
end SamplingTasks;
```

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

```
package Redundancy
```

```
public
```

```
    abstract Triple
```

```
    features
```

```
        input: in data port;
```

```

    output: out data port;
end Triple;

abstract implementation Triple.impl
subcomponents
    MyProcess: abstract Calculate [3];
    MyVoter: abstract Voter;
connections
    extinput: port input -> MyProcess.input
                { Connection_Pattern => (( One_To_All )); };
    tovoter: port MyProcess.output -> MyVoter.input
                { Connection_Pattern => (( One_To_One )); };
    extoutput: port MyVoter.output -> output;
end Triple.impl;

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

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

```

4.6 Abstract Components

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

Legality Rules

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

- (L1) An **abstract** component type declaration can contain feature declarations (including abstract feature declarations), flow declarations, as well as property associations.
- (L2) An **abstract** component implementation can contain subcomponent declarations of any category. Certain combinations of subcomponent categories are only acceptable if they are acceptable in one of the concrete component categories.
- (L3) An **abstract** component implementation can contain a modes subclause, a connections subclause, a flows subclause, and property associations.
- (L4) An **abstract** subcomponent can be contained in the implementation of any component category.
- (L5) If an **abstract** subcomponent is refined to a concrete category, the concrete category must be acceptable to the component implementation category whose subcomponent is being refined.
- (L6) An **abstract** subcomponent can be declared as an array of subcomponents.
- (L7) If an **abstract** component type is refined to a concrete category, the features, modes, and flow specifications of the abstract component type must be acceptable for the concrete component type.
- (L8) If an **abstract** component implementation is refined to a concrete category, the subcomponents, call sequences, modes, flow implementations, and end-to-end flows of the abstract component implementation must be acceptable for the concrete component implementation.

Standard Properties

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

Semantics

- (3) The component of category **abstract** represents an abstract component. It can be used to represent conceptual architectures. This abstract component can be refined into a runtime architecture by refining the component