|  |  |  |
| --- | --- | --- |
| **AEROSPACE** **standard** | as5506™ | REV. D |
| Draft 2020-01  Superseding AS5506C | |
| Architecture Analysis and Design Language (AADL): Part 1 | | |

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 and V2.2, revisions that address a number of errata and minor improvements agreed upon by the committee, were published as AS5506B in 2012 and AS5506C in 2017.

This document AS5506D documents AADL V3, a major revision AADL based on industrial experience, using AADL V2.2 as baseline. This revision introduces new concepts in addition to addressing errata.

Part 1 of the AADL standard document introduces general architecture description concepts.

Notes:

The 2020-01 draft has the following additions and revisions:

* State behavior expressed through state variables and state transitions,
* Use of state variables to represent modes and mode-specific architecture topology
* Component behavior – a placeholder to use the AADL V3 expression language and the AADL V3 state behavior to express the capabilities of the Behavior Annex,
* Typed token behavior to model tpe token systems including the ability to express the capabilities of the Error Model V2 (EMV2) Annex,
* Annotations to associate model information rather than system properties with a model.

Table of Contents

[1. Packages 3](#_Toc29925423)

[2. Classifiers 5](#_Toc29925424)

[3. Interfaces 5](#_Toc29925425)

[4. Implementations 8](#_Toc29925426)

[5. Subcomponents 11](#_Toc29925427)

[6. Configurations 13](#_Toc29925428)

[7. Classifier Assignments 15](#_Toc29925429)

[8. Features 17](#_Toc29925430)

[8.1 Ports 18](#_Toc29925431)

[8.2 Access Features 18](#_Toc29925432)

[8.3 Abstract Features 19](#_Toc29925433)

[8.4 Subprogram Parameters 19](#_Toc29925434)

[8.5 Named Interfaces 20](#_Toc29925435)

[9. Component Relationships 20](#_Toc29925436)

[9.1 Connections 21](#_Toc29925437)

[9.2 Deployment Bindings 22](#_Toc29925438)

[9.2.1 Binding Types 23](#_Toc29925439)

[9.2.2 Bindings 23](#_Toc29925440)

[9.2.3 Binding Points 24](#_Toc29925441)

[9.2.4 Qualified and Quantified Resources 24](#_Toc29925442)

[10. Component Flows 24](#_Toc29925443)

[10.1 Flow Specifications 25](#_Toc29925444)

[10.2 Flow Paths and Flow Sequences 26](#_Toc29925445)

[11. State Behavior 28](#_Toc29925446)

[11.1 State Variables 28](#_Toc29925447)

[11.2 State Transitions 28](#_Toc29925448)

[11.3 Generators 29](#_Toc29925449)

[12. Modes and Mode-Specific Architecture Topology 30](#_Toc29925450)

[13. Component Behavior 31](#_Toc29925451)

[14. Typed Token Behavior 31](#_Toc29925452)

[14.1 Simple Token Propagation 32](#_Toc29925453)

[14.2 Token Propagation and Transformation 33](#_Toc29925454)

[14.3 Stateful Typed Token Propagation 33](#_Toc29925455)

[14.4 Annotated Type Token Systems 34](#_Toc29925456)

[14.5 Error Behavior Specification 34](#_Toc29925457)

[15. Annotations 37](#_Toc29925458)

[16. Annex Subclauses and Annex Libraries 37](#_Toc29925459)

**No table of figures entries found.**

# Packages

Description

1. A *package* provides a way to organize definitions of classifiers, data types, properties, and property sets – referred to as *package elements*. Packages also represent annex libraries, i.e., collections of annex sublanguage specific definitions (see Section 9). Packages can be syntactically nested.
2. A collection of packages makes up an AADL specification.
3. Packages are referenced by their name. In case of syntactically nested packages this includes the name of enclosing packages.
4. Package elements and nested packages can be marked as private to indicate that they not externally accessible. This means that they cannot be referenced from outside the package.
5. Package elements within the same package can be referenced by their name. Package elements contained in other packages can be referenced by their name if the package content is made visible by an import declaration.
6. An imported definition can have an *alias* which allows users to resolve conflicts in imported names.
7. Package element references can always be qualified by a package name. This allows users to refer to imported definitions that have the same name as local definitions or other imported definitions. In this case the referenced definition does not have to be made visible through an import declaration.

Syntax

AADLSpecification ::= { Package }+

Package ::=

{ Annotation }\*

**package** PackageName **is**

{ ImportDeclaration | PackageElement | NestedPackage }\*

**end ;**

PackageName ::= Identifier { **::** Identifier }\*

PackageElement ::=

{ Annotation }\*

[ **private** ] ( DataType | Classifier | Property | PropertySet )

NestedPackage ::=

[ **private** ] Package

ImportDeclaration ::= **import** ImportReference **;**

ImportReference ::= WildcardPackageReference

| PackageElementReference [ Alias ]

Alias ::= **as** Identifier

WildcardPackageReference ::= PackageName**::\***

PackageElementReference ::=

[ PackageName **::** ] ( ClassifierName | DataTypeName | PropertyName | PropertysetName )

Naming Rules

1. Package names reside in a global name space. For a syntactically nested package, the package name is prefixed with the package name of enclosing the packages.
2. A package provides a name space for definitions contained in the package.
3. The package name in a package element reference or a wildcard package reference must resolve in the global name space.
4. Each package introduces a namespace for its package elements.
5. An import reference to a package with a wild card <star> makes all of its non-private package elements visible to the package containing the import declaration.
6. An import reference to a package element makes the referenced definition visible to the package containing the import declaration.
7. The alias identifier resides in the name space of the package containing the import declaration.
8. A reference to a package element without qualifying package name is resolved in the following order:

* as name to a definition in the name space of the package containing the reference,
* as identifier of an alias, which resolves to the qualified references of the package element in the import declaration,
* as name to a package element made visible by an import declaration.

Legality Rules

1. A package element marked as *private* must only be referenced within the package it is defined in.
2. A reference qualified by package name does not have to be made visible through an import declaration.

Processing Requirements and Permissions

1. A method of processing must accept an AADL specification presented as a single string of text in which packages 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.
2. Top-level packages, i.e., packages that are not syntactically nested, may be stored in separate files.
3. 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.

Examples

1. Two packages with nested names. The first contains a syntactically nested package. The second makes the content of the first package visible via import.

**package** PackP::Q **is**

**type** date ;

**type** signal;

**interface** nested **is**

signal: **feature**;

**end;**

**end;**

**package** PackP **is**

**import** PackP::Q::\*;

**thread** **interface** mythread **is**

today: **in port** date;

**end;**

**interface** mynested **extends** nested **is**

p1: **feature** signal;

**end;**

**end;**

# Classifiers

*Description*

1. A *classifier* represents a description of a component. This can be the interface, implementation, or configuration definition of a component.
2. A classifier represents a component of a specific *component category*. The semantics of each component category are defined in Part 2 and Part 3 of this document.
3. A classifier reference can be qualified with a package name.
4. Elements contained in classifiers include definitions of model elements. The specific kinds of model element definitions as well as other declarations are described in the sections for interfaces (Section 3), implementations (Section 4), and configurations (Section 6).

Syntax

Classifier ::= Interface | Implementation | Configuration

ClassifierName ::= InterfaceName | ImplementationName | ConfigurationName

ClassifierReference ::= [ PackageName **::** ] ClassifierName

ModelElementReference ::= Identifier { **.** Identifier }\*

*Naming Rules*

1. A classifier introduces a local name space for model elements defined within the classifier.
2. A classifier may be defined as an extension of another classifier. In this case the extension inherits the name space of the classifier(s) being extended.
3. A classifier reference is resolved according to the naming rules for package element references.
4. A model element reference is resolved as follows:

* The first identifier is resolved in the name space of the classifier containing the reference,
* The next identifier is resolved in the name space of the model element identified by the previous identifier.

# Interfaces

*Description*

1. An *interface* represents an abstraction of a component. The component interface provides a contract for how external components interact with the component. All such interactions are limited to occur through the component features.
2. An interface can be specified for a specific component category or without a category. In the latter case the interface can be combined with other interfaces as part of a composition.
3. An interface includes:

* *features* to represent interaction points with other components with some features representing directional interaction,
* *binding points* to represent endpoints for deployment binding of application components to execution platform components,
* *generators* as originators of event, data streams, or typed token streams,
* *flow specifications* indicating flow sources, sinks, and paths from incoming to outgoing features including computational mappings of values,
* *state behavior* specifications in the form of state variables and transitions indicating externally visible operational modes,
* *typed token behavior s*pecifications in the form of typed token pass thru and transformation based on coming token conditions and on state conditions,
* *computational component behavior* specifications in the form of input processing to produce output possibly maintaining state,
* *annex subclauses* that specify additional characteristics of the component,
* *associations of property value*s to the component interface and its elements, and
* *classifier assignments* to associate types with features of interfaces being extended.

1. Interface extension allows users to represent related systems and provide libraries of reusable interface definitions.
2. An interface can be defined as extension of other interfaces. An extension inherits all elements of the interfaces being extended. An extension can add new elements and assign a data type or classifier to an inherited feature.
3. An interface extension of multiple interfaces combines all elements of the interfaces become part of the resulting composite interface.
4. The same interface can be included multiple times through the use of named interface definitions. Similarly, different component interfaces with conflicting features can be combined through the use of named interface declarations.
5. The direction of directional features in an interface being extended is reversed if the keyword reverse is specified. This is useful when an interface is used in interface compositions for a sender component and a receiver component using the same interface definition.
6. An interface can contain a named interface as feature, which allows for nesting of interfaces, i.e., collections of features. For details see section 7.8.

*Syntax*

Interface ::=

[ Category ] **interface** InterfaceName

[ **extends** [ **reverse** ] InterfaceReference { **,** [ **reverse** ] InterfaceReference }\* ]

**is** { ( { Annotation }\*InterfaceElement ) | *InterfaceElement*AnnotationBlock}\* **end;**

InterfaceName ::= Identifier

Category ::=

GenericCategory | SoftwareCategory | ExecutionPlatformCategory | CompositeCategory

GenericCategory ::= **abstract**

SoftwareCategory ::=

**thread** | **thread group** | **process** | **data** | **subprogram** | **subprogram group**

ExecutionPlatformCategory ::=

**memory** | **processor** | **bus** | **device** | **virtual processor** | **virtual bus** | **virtual memory**

CompositeCategory ::= **system**

InterfaceReference ::= [ PackageName **::** ] InterfaceName

InterfaceElement ::=

Feature | BindingPoint | Behavior | StateVariable | Transition | Generator

| AnnexSubclause | PropertyAssociation | ConfigurationAssignment

*Naming Rules*

1. Features, binding points, flow specifications, and mode specifications are model elements whose defining names reside in the local name space of the interface.
2. An interface extension inherits the name space of the interfaces being extended. This means there cannot be two definitions with the same name in different interfaces being extended or a definition added in the extension.
3. An interface reference is resolved according to the naming rules for package element references.

*Legality Rules*

1. The category of the interface definition must be the same as the category of any interface being extended, or the interface being extended must have been defined without a category.
2. The category of an interface must not be *data*.
3. There must only be one property association for the same property in any of the interfaces being extended.
4. Classifier assignments must only be declared in interface extensions.
5. There must only be one classifier assignment for the same feature in any of the interfaces being extended.

Examples

**package** InterfaceComposition **is**

**interface** Logical **is**

temperature : **out** **port** ;

Speed : **out** **port** ;

**end**;

**interface** Physical **is**

Network : **requires** **bus** **access** CANBus;

**end**;

-- Add port in extension

**interface** s1 **extends** Logical **is**

Onemore : **out** **port** ;

**end**;

-- combine two interfaces

**thread interface** sender **extends** Logical , Physical

**end**;

-- combine two interfaces with one having direction reversed

**thread interface** receiver **extends** **reverse** Logical , Physical

**end** ;

-- combine two interfaces and locally add a feature

**interface** s3 **extends** Logical , Physical **is**

Onemore : **out** **port** ;

**end** ;

**bus** **interface** CANBus **is end**;

**end**;

# Implementations

Description

1. An *implementation* represents the realization of a component that satisfies the interface identified by the implementation name. An implementation provides the details of how a component is implemented through subcomponents, their internal connections and connections to features in the interface.
2. This leads to a component hierarchy for system architectures, hardware platform architectures, and software task and communication architectures. Functional code is represented as an abstraction.
3. An implementation consists of

* *subcomponents* that represent instances of component inside a given component. Subcomponents may themselves contain subcomponents leading to a component hierarchy,
* *connections* that represent interactions between subcomponents and delegation of connection end points between an enclosing component and a subcomponent,
* *bindings* that represent deployment of application components to platform components,
* *generators* as originators of event, data streams, or typed token streams,
* *flow behavior specifications* indicating flow sources, sinks, and paths from incoming to outgoing features including computational mappings of values,
* *flow sequences* that represent implementations of flow specifications in the component interface, and end-to-end flows with starting and end points within the component implementation,
* *state behavior specifications* in the form of state variables and transitions,
* *typed token behavior s*pecifications in the form of typed token pass thru and transformation based on coming token conditions and on state conditions,
* *computational component behavior* specifications in the form of input processing to produce output possibly maintaining state,
* *annex subclauses* that specify additional characteristics of the component,
* *associations of property values* specific to the implementation and its contained elements, and
* *classifier assignments* to assign classifiers or data types to configure subcomponents of extended implementations and features of extended interfaces.

1. An implementation extension can add implementation elements to an existing implementation. Furthermore, implementation extensions can refine existing subcomponents and features through classifier assignments and property associations.
2. An interface can have multiple implementations. An implementation can be viewed as a component variant with differing property values that characterize the differences between implementations. Users can select and configure specific implementations for subcomponents.

Syntax

Implementation ::=

Category ImplementationName

[ **extends** ImplementationReference ]

**is** { ({ Annotation }\*ImplementationElement) | *InterfaceElement*AnnotationBlock}\* **end**;

ImplementationName ::= Identifier **.** Identifier

ImplementationReference ::=

[ PackageName **::** ] ImplementationName

ImplementationElement ::=

Subcomponent | Connection | Binding | FlowSequence | Behavior |

StateVariable | Transition | Generator |

AnnexSubclause | PropertyAssociation | ConfigurationAssignment

Naming Rules

1. The first identifier of an implementation name identifies the interface the implementation is associated with. The interface name must exist in the name space of the package containing the implementation definition or it must be visible through an import declaration.
2. Implementation elements with defining identifiers are model elements whose defining names reside in the local name space of the implementation.
3. An implementation inherits the name space of its associated interface.
4. An implementation extension inherits the name space of the implementation being extended.
5. An implementation reference is resolved according to the package element reference naming rules.

Legality Rules

1. An implementation must have a category.
2. The category of the implementation must be the same as the category of the interface it is associated with or the interface must not have a category.
3. The category of an implementation extension must be the same as the category of the implementation being extended.
4. Classifier assignments must only be declared in implementation extensions.

Processing Requirements and Permissions

1. An implementation may denote a set of actual system components using references to source code.(i.e., by identifying source code files), existing or potential. Existing, actual components should be verified to be compliant with their AADL implementation definition as well as their associated interface.
2. When component implementations denoted by two or more different actual implementations, and the actual implementations should differ (i.e., it is intended that the implementations should differ in details such as internal structure or behaviors), these intended differences may be specified using properties or annex subclauses.
3. Some differences between implementations are not architecturally visible, nor should they be controlled at an architectural level of specification. In general, two actual components (i.e., realized in source code) that comply with the same interface and implementation are not necessarily substitutable for each other in an actual system. This is because an AADL specification may be legal but not specify all of the characteristics that are required to ensure total correctness of a final assembled system. For example, two different versions of a piece of source text might both comply with the same AADL specification, yet one of them may contain a programming defect that results in unacceptable runtime behavior. Compliance with this standard alone is not sufficient to guarantee overall correctness of an actual system.

Examples

**package** ConnectionExample **is**

// Functional architecture using abstract components

**type** systemstate;

// The sensor functional interface specifies interactions with the physical environment

// through a feature and provision of readings to other components.

// It shares access to a data component representing some application state.

**abstract**  **interface** sensor **is**

state: **requires** **in** **data** **access** systemstate;

outp: **out** **port**;

fea: **feature**;

**end**;

// A sensor implementation specifies that the component internally uses a thread.

**abstract** sensor.i **is**

sub1: **thread** sense;

upmap: **connection** sub1.p1 -> outp ;

stateaccess: **connection** state -> sub1.state;

**end**;

// An actuator functional interface specifies input for another component which

// result in effects on a physical component through the feature.

**abstract**  **interface** actuator **is**

state: **provides** **in** **data** **access** systemstate;

inp: **in** **port**;

fea: **feature**;

**end**;

// A sensor implementation specifies that the component internally uses a thread.

**abstract** actuator.i **is**

mystate: **data** systemstate ;

sub2: **thread** actuate;

downmap: **connection** inp -> sub2.p1;

stateread: **connection** mystate -> state ;

statewrite: **connection** sub2.state -> mystate ;

**end**;

**end**;

# Subcomponents

Description

1. A *subcomponent* is instantiated when the containing implementation is instantiated. If the subcomponent declaration references an implementation or configuration, its subcomponents are instantiated recursively. This results in a component instance hierarchy.
2. Property values can be associated with subcomponents by declaring them in curly brackets as part of the subcomponent declaration. Property values can also be associated by a property association declaration that references the subcomponent.
3. Subcomponents including their property values can be configured through classifier assignments in implementation extensions or configurations.
4. *Nested subcomponents* can be declared as part of a subcomponent declaration inside curly brackets. This allows users to define a subcomponent hierarchy without explicitly defining classifiers.
5. Subcomponents can be declared when a state condition holds, e.g., to represent that a component is active in a certain set of modes.

Syntax

Subcomponent ::=

Identifier **:** Category ( TypedSubcomponent | ( [ InModes ] NestedSubcomponent ) )**;**

TypedSubcomponent ::=

TypeReference [ InModes ] [ **{** { PropertyAssociation }+ **}** ] [ **;** ]

TypeReference ::= ClassifierReference | DataTypeReference

NestedSubcomponent ::=

**{** { { Annotation }\*NestedElement ) | *NestedElement*AnnotationBlock }+ **}**

NestedElement ::= PropertyAssociation | Subcomponent | Feature | Connection

SubcomponentReference ::= Identifier

Legality Rules

1. The category of the referenced classifier must be the same as the category of the subcomponent definition, or it may reference an interface without category.
2. If the category of a subcomponent declaration is of category *data*, then it must reference a data type.
3. The classifier of a subcomponent cannot recursively contain subcomponents with the same classifier. In other words, there cannot be a cyclic containment dependency between components.

Processing Requirements and Permissions

1. If the subcomponent references an interface and the interface 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 interface has multiple implementations then the implementation must be explicitly identified. However, some project may impose design constraints that require modelers to completely specify such classifier references.

Examples

1. The example below illustrates the use of nested subcomponents. It shows an alternative means of instantiation that can be compared to the example in the previous section, which illustrated the use of classifier-based subcomponent definitions and instantiation.
2. In this example of nested subcomponents, curly braces are used to specify subcomponents inline without an explicit classifier. The subcomponents are of the following list of AADL component categories: a “sensing” **device**, a “processing” **process** with a “control” **thread**, and an “actuating” **device**.

**package** NestedImplementations **is**

**import** StandardProperties::\*;

**system** **interface** ControlSystem **is end**;

**system** ControlSystem.i **is**

sensing : **device** {

sensedata : **out** **port** ;

} ;

processing : **process** {

inp: **in** **port**;

filter : **thread** {

inp : **in** **port** ;

outp : **out** **port** ;

} ;

control : **thread** { #Period => 22 ms ;

inp : **in** **port** ;

outp : **out** **port** ;

inp#Data\_Size => 45 Bytes;

} ;

filtercontrolconn : **connection** filter.outp -> control.inp ;

outp: **out** **port**;

outmap: **connection** control.outp -> outp ;

outp#Data\_Size => 45;

} ;

actuating : **device** {

inp : **in** **port** ;

} ;

sensefilterconn : **connection** sensing.sensedata -> processing.filter.inp ;

controlactuateconn : **connection** processing.outp -> actuating.inp;

reachdowncontrolactuateconn : **connection** processing.control.outp -> actuating.inp;

**end**;

**end**;

# Configurations

Description

1. A *configuration* is defined for an interface, implementation, or other configurations. A configuration allows users to expand the leaves of a component hierarchy without modifying the existing topology by assigning an implementation to subcomponents that have been declared with an interface. Users can also add model elements such as property values, flows, behavior specifications, to the existing component and connection topology, but cannot change the topology by replacing implantations specified for subcomponents, or by adding subcomponents or connections.
2. A configuration consists of a collection of any of the following:

* *classifier assignments* to assign implementations or configurations to subcomponents in the component hierarchy. It can also assign a data type or component interface to features of components in the component hierarchy
* *bindings* that represent deployment of application software components to platform hardware components,
* *flow specifications* for flows within components by mapping incoming features to outgoing features and *flow sequences* across components,
* *annex subclauses* that specify additional characteristics for model elements,
* *property associations* to configure property values for components and other model elements.

1. The interface or implementation being configured may be explicitly specified after the *extends* reserved word or it is inferred from the configuration(s) being extended.
2. A configuration can represent a composition of configurations listed after the *extends* reserved word. In this case the implementation of the composite configuration is the furthest descendent in a single extends lineage of the configurations being composed.
3. A configuration can be parameterized. In this case the component hierarchy represented by the subcomponents can only be configured through the parameters.
4. A configuration can be defined to contain only generally applicable classifier assignment patterns without being an extension of a classifier. In this case it is defined with a single identifier. When assigned to a subcomponent, this component becomes the root for applying the classifier assignment patterns of such a configuration.

NOTE: This is effectively an implementation extension with the restriction that the extension does not contain subcomponent and connection declarations or subcomponent refinements that replace one implementation by a different one. Different from AADL V2 subcomponent refinements, classifier assignments can reach down the component hierarchy to assign a classifier to a subcomponent.

Syntax

Configuration ::=

**configuration** ConfigurationName [ ConfigurationParameters ]

**extends** ClassifierReference { **,** ClassifierReference }\*

**is** { ( { Annotation }\*ConfigurationElement ) | *ConfigurationElement*AnnotationBlock }+

**end ;**

ConfigurationName ::= Identifier [ **.** Identifier ]

ConfigurationReference ::=

[ PackageName **::** ] ConfigurationName

[ **(** ConfigurationActual { **,** ConfigurationActual }\* **)** ]

ConfigurationParameters ::=

**(** ConfigurationParameter { **,** ConfigurationParameter }\* **)**

ConfigurationParameter ::= Identifier **:** TypeReference

ConfigurationElement ::=

ConfigurationAssignment | PropertyAssociation | ModeAssignment

| ConfigurationAssignmentPattern

Naming Rules

1. If the configuration name consists of two identifiers, the first identifier must be the same as that of the interface, implementation, or configurations being extended.
2. A configuration defines a local name space for the defining identifiers of its configuration parameters. This name space inherits the name space of interface, implementation, and configurations being extended.
3. Model element references that include subcomponents with parameterized configurations can only refer to parameters of the configuration, but not to model elements whose definition is inherited from classifiers being extended.

Legality Rules

1. A configuration reference must only include parameter actuals if the configuration is defined as parameterized.
2. If a configuration extends multiple classifiers then at most one classifier reference must be to an interface or to an implementation.
3. If a configuration extends multiple classifiers then the interface of these classifiers must be the same.
4. If a configuration extends multiple classifiers then the implementations of these classifiers must have a single extends lineage, i.e., the implementations must be the same or they must be ancestors of one implementation.
5. A configuration defined with a single identifier must only contain classifier assignment patterns.
6. A configuration defined with a single identifier is not associated with an interface or implementation, i.e., rules (L3) and (L4) do not apply when it is referenced as being extended.

# Classifier Assignments

Description

1. A *classifier assignment* assigns one or more configurations and at most one component implementation to a specific subcomponent the component hierarchy that serves as the starting point for further elaboration within the component hierarchy. If the subcomponent category is data a data type is assigned. If the category is abstract the assignment can be an implementation, configuration(s), or a data type.
2. Multiple classifier assignments can be made to the same subcomponent in the component hierarchy. They may be declared in the same implementation extension or configuration or in different implementation extensions configurations assigned to the subcomponent or an enclosing subcomponent.
3. Assigned configurations may add bindings, flow specifications and flow sequences, property associations, and classifier assignments to components further down the hierarchy, with respect to the specific subcomponent that serves as the starting point within the component hierarchy.
4. Classifier assignments declared in implementation extensions are subcomponent refinements. In that case a classifier assignment can replace the interface reference of a subcomponent by an implementation associated with the interface, and replace an implementation reference by an implementation extension. If the target of the classifier assignment is a subcomponent defined directly in the implementation of the configuration, then the interface of the assigned classifier(s) can be an interface extension.
5. Classifier assignments declared in configurations extend but do not change the existing topology of subcomponents and connections. In that case the interface of a subcomponent definition or refinement by an implementation extension must not change. In other words the classifiers being configured in cannot extend the interface of the subcomponent definition or its refinements and the implementation associated with the assigned classifiers must be the same or an ancestor of the implementation of the subcomponent definition or refinement. If the subcomponent only specifies an interface then one of the assigned classifiers must identify an implementation and implementations associated with other assigned classifiers must be the same or an ancestor.
6. A classifier assignment can assign an appropriate data type or interface to a feature of a subcomponent in the component hierarchy. It may override the type reference of the feature if present.
7. A *classifier* *assignment pattern* allows users to perform a classifier assignment to all model elements whose classifier is the same or an extension of the classifier specified in the pattern.

Syntax

ClassifierAssignment ::=

ModelElementReference **=>**

( ( ConfigurationValue [ Subconfiguration ] [ **;** ] )

| Subconfiguration ) **;**

ClassifierValue ::=

DataTypeReference | ImplementationReference | ConfigurationReference

| ConfigurationParameterReference

ConfigurationParameterReference ::= Identifier

Subconfiguration ::= **{** { ConfigurationElement }+ **}**

ConfigurationAssignmentPattern ::=

**all(**ClassifierReference**)** **=>**

( ( ConfigurationValue [ Subconfiguration ] [ **;** ] )

| Subconfiguration ) **;**

Naming Rules

1. The model element reference of classifier assignments is resolved in the context of the classifier that contains the classifier assignment. The sequence of identifiers represents a path to a subcomponent through the subcomponent hierarchy or a path to a feature, possibly contained in a named interface, of the enclosing classifier or of a subcomponent identified by the first part of the path. However, the path cannot reach into a subcomponent with a parameterized configuration.
2. In the case of classifier assignments declared in subconfigurations the model element reference is resolved with respect to the enclosing subcomponent being configured.
3. The configuration parameter reference must identify a configuration parameter of the configuration that contains the classifier assignment.

Legality Rules

1. The model element reference must identify a subcomponent or feature in the component hierarchy of the component classifier that contains the classifier assignment.
2. For a subcomponent model element reference of a classifier assignment whose target is a data component the assigned value must be a data type.
3. For feature model element references to ports, data access features, or abstract features, the assigned configuration value must be a data type. For other access features it must be a compatible component interface.
4. If the assigned configuration is parameterized then parameter actuals must be included.
5. If one of the assigned classifiers is a parameterized configuration, then additional configurations are limited to assigning missing property values, flows, and annex specifications.
6. If the assigned classifier is a configuration with a single identifier then it can be applied to any component.

The next legality rules apply to classifier assignments declared in implementation extensions.

1. The interface of the classifier(s) being assigned must be for the same or an extension of the interface of the subcomponent definition or refinement.
2. If the subcomponent being configured references an implementation or configuration in its definition or a refinement, then the implementation of the classifier(s) being assigned must be the same, an ancestor, or an extension of the subcomponent implementation or configuration’s implementation. In the case of multiple classifiers the classifier implementations must form a single extends lineage.

The next legality rules apply to classifier assignments declared in configurations.

1. The interface of the classifier(s) being assigned must be the same as the interface of the subcomponent referenced in its definition or refinement.
2. If the subcomponent being configured references an implementation or configuration in its definition or refinement, then the implementation of the classifier(s) being assigned must be the same, or an ancestor of the subcomponent implementation.
3. If the subcomponent being configured references an interface in its definition or refinement, then one of the assigned classifiers must be an implementation. All other classifiers assigned through configurations must have an associated implementation that is the same or an ancestor of the specified implementation.

Examples

1. Add an example.

# Features

Description

1. A *feature* represents an interaction point of a component with external components that is defined as part of tis interface. All interactions of subcomponents of any component implementation must go through a feature.
2. Features may be specified without direction (non-directional) or they may indicate a direction. Incoming features are those with the keyword in or in out. Outgoing feature are those with the keyword out or in out. Feature with the keyword in out are considered bi-directional.
3. AADL models may use the following categories of features:

* *ports* represent directional flow of messages with or without data. Their arrival can trigger a dispatch or mode transition of the receiving component.
* *access features* represent coordinated access to shared components of categories *data*, *subprogram*, *subprogram group*, *bus*, *virtual bus*.
* subprogram *parameters* represent parameters of subprograms.
* *abstract features* represent general interaction points without software or hardware specific interaction semantics.
* *named interfaces* represent collections of features of an interface that can be referenced by their named interface identifier.

1. The type reference of a feature is optional and can be configured through a classifier assignment.

Syntax

Feature ::=

FeatureName **:** [ FlowDirection ]

( PortFeature | AccessFeature | AbstractFeature | NamedInterface )

[ **{** { PropertyAssociation }+ **}** ] **;**

FeatureName ::= Identifier

FlowDirection ::= **in**  |  **out**  | **in out**

Naming Rules

1. A feature name is part of the local name space of a component interface or of a named interface feature.

Legality Rules

1. Each component category section specifies which feature categories are legal for that component.

## Ports

Description

1. A *port* represents an interaction point for discrete directional message communication between components along connections.
2. Messages can be data with a specified data type or represent events without additional information, by referencing a predefined type called event. The data type can be an enumeration type to communicate literals, or it can be *type* indicating that a type is communicated, e.g., a type representing an error type.

Syntax

PortFeature ::= **port** [ DataTypeReference ]

Legality Rules

1. The port feature must have a flow direction.
2. The optional keyword *event*must only be present on incoming ports.

Examples

**thread** **interface** filter **is**

rawReading: **in** **port** SensorReading;

filteredReading: **out** **port** FilteredReading;

**end**;

## Access Features

Description

1. An *access feature* is used to model access to shared components. The access may be directional, bi-directional, or non-directional.
2. Requires indicates that the component with the access feature requires access to an external component. Provides indicates that the component with the access feature provides access of an internal component to external components.
3. For access direction without the optional flow direction the keyword provides represents outgoing while the keyword requires represents incoming.
4. The combination provides out and requires in represent read access, while requires out and provides in represents write access. Requires in out and provides in out represent read/write access.
5. The type reference indicates the type of the component to be accessed.

Syntax

AccessFeature ::= AccessDirection AccessCategory **access** [ TypeReference ]

AccessDirection ::= ( **requires**  |  **provides** )

AccessCategory ::= **data** | **bus** | **virtual bus** | **subprogram** | **subprogram group**

Examples

**thread** **interface** sense **is**

state: **requires** **in** **data** **access** systemstate;

p1: **out** **port** ;

**end**;

## Abstract Features

Description

1. An *abstract feature* represents generic features without specific software related interaction semantics. It is often used in conceptual, functional, and physical system models.
2. A data type may indicate the type of interaction, e.g., the type of resource being exchanged such as electricity.

Syntax

AbstractFeature ::= **feature** [ DataTypeReference ]

Examples

**device** **interface** sensor **is**

air: **in** **feature** Air;

temperatureReading: **out** **port** Temperature;

**end**;

## Subprogram Parameters

Description

1. A *subprogram parameter* represents the parameters of a subprogram.

Syntax

SubprogramParameterFeature ::= **parameter** [ DataTypeReference ]

Legality Rules

1. The subprogram parameter feature must have a flow direction.

## Named Interfaces

Description

1. A *named interface* defines a feature that represents a collection of features specified by another interface.
2. This allows users to define interfaces as composition of multiple instances of the same interface.
3. Named interfaces can also be used to address name conflicts between features of different interfaces listed after the *extends*.
4. Named interfaces are considered to be bi-directional as their elements may be incoming or outgoing.
5. Reverse indicates that incoming features in the referenced interface are considered to be outgoing and vice versa.
6. Note: Named interfaces replace feature groups in AADL V2, and interfaces replace feature group types.

Syntax

NamedInterface ::= **interface** [ **reverse** ]

Naming Rules

1. The named interface inherits the name space of the referenced interface definition.
2. A named interface element is referenced by a model element path that ends with the named interface identifier followed by the named interface element.

Legality Rules

1. The named interface must not have a flow direction.
2. The type reference must refer to an interface definition.

Examples

**interface** SenseFunction **is**

Network : **requires** **bus** **access** CANBus;

**end**;

**interface** PhysicalInterface **is**

CANBus : **requires bus access** CANBus ;

**end**;

**device** **interface** sensor **is**

Function: **interface** SensorFunction;

Physical: **interface** PhysicalInterface ;

**end**;

# Component Relationships

1. The following types of component relationships are supported:

* *connections* to represent interactions between subcomponents as well as delegation of connection end points from an enclosing component features to subcomponent features,
* *flow specifications* to represent flows between incoming and outgoing features as abstraction for flows within components,
* *flow sequences* to represent flows within components as sequences of alternating connections and subcomponent flow specifications.

1. Flow specifications and flow sequences can be defined for any component, i.e., users can specify flows through functional and physical architectures, as well as software and hardware platform architectures.

## Connections

Description

1. A *connection* specifies an interaction between two subcomponents through one of their features of the same feature category, the connection end points.
2. A connection also specifies feature delegation of connection end points, i.e., how a feature of an enclosing component that is a connection end point is delegated to a feature of a subcomponent. Such connection definitions follow the flow direction, i.e., for incoming features the enclosing component feature is specified first, while for outgoing features the enclosing component feature is specified second.
3. A connection instance is determined by a connection with the endpoints expanded down the component hierarchy according to specified feature delegations. More than one feature delegation for an endpoint results in separate connection instances. More than one feature delegation for both endpoints results in a separate connection instance of each combination of delegations.
4. A connection between named interfaces represents a collection of connections between the features contained in the referenced interface. Each of these connections results in connection instances with the endpoints expanded according to feature delegations.
5. A connection is directional (**->**) or bi-directional (**<->**). In case of directional connections, the direction is from an outgoing feature to an incoming feature. In case of feature delegation, it follows outgoing features up the component hierarchy and incoming features down the component hierarchy. In the case of bi-directional connections all connection and feature delegation endpoints must be bi-directional or non-directional.
6. For connections involving access features the provides access feature can be replaced by the subcomponent being accessed. Furthermore, a component through component can only access a single access feature.
7. Connections and feature delegations can reach into named interface features, i.e., they can refer to features inside named interfaces.
8. Connections and feature delegations can reach down the subcomponent hierarchy if the subcomponent is defined as nested subcomponent and without features.
9. Connections can be declared when a state condition holds, e.g., to represent that a connection is active in a certain set of modes.

Syntax

Connection ::=

Identifier **: connection** ModelElementReference (  **->** | **<->** ) ModelElementReference

[ InModes ] [ **{** { PropertyAssociation }+ **}** ]**;**

Naming Rules

1. Model element references are resolved in the name space of the implementation that contains the connection definition with subsequent identifiers being resolved in the name space of the preceding model element.

Legality Rules

1. For connections between subcomponents both model element references must start with a subcomponent and identify a feature possibly nested inside named interface features through a sequence of one or more feature identifiers. In case of connections involving access features one of the model element references may identify the subcomponent to be accessed instead of an access feature.
2. For feature delegation one of the model references must not start with a subcomponent and identify a feature possibly nested inside named interface features through a sequence of one or more feature identifiers. In case of feature delegations involving access features the reference starting with a subcomponent may identify the subcomponent to be accessed instead of an access feature.
3. Either model element reference may start with a sequence of more than one subcomponent identifiers if the identified subcomponent is defined as a nested subcomponent and the nested subcomponent does not include feature definitions.
4. The feature category of both referenced features must be the same. In the case of an access feature the access feature category must match the subcomponent category. In the case of a parameter feature reference the other reference must be to a parameter or port.
5. The type reference of both referenced features or referenced feature and subcomponent must be the same.
6. For directional connection definitions ( **->** ) the first model element reference must identify an outgoing feature and the second model element reference must identify an incoming feature or instead of incoming access features a subcomponent.
7. For bi-directional connection definitions ( **<->** ) both model element references must identify a bi-directional or non-directional feature, or for access features one model element reference may identify a subcomponent..

## Deployment Bindings

Description

1. Deployment bindings associate components of one architecture layer to components of another layer. For example, bindings are used to map components of a functional architecture to components of a physical architecture, or component of a software application architecture to components of a virtual or physical hardware platform.
2. Bindings are characterized by a *binding type*. The binding type identifies to component categories of the source and target components.
3. A *binding* deploys a source component to a target component. The source and target must be consistent with the binding type.
4. Users may define a *binding point* in the interface of a specific component that can become the source or target of a binding. See section 8.6 for details.
5. Bindings may involve resources. A component that is a binding target may provide multiple resource types, e.g., a processor may provide processing cycles and a particular instruction set. Users may bind to each resource separately via binding points.
6. The resource characteristics are represented by qualifying or quantifying properties indicating the resource required by the binding source and provided by the binding target.

### Binding Types

Description

1. A *binding type* is used to distinguish between different types of deployment bindings.
2. A binding type identifies the component categories of the source and target components.
3. A binding type is defined through the type system. It may/must specify a map between to unions of types.
4. Several binding types are predefined:

* *FunctionalBinding* to bind elements of a functional architecture to elements of a physical architecture.
* *ThreadBinding* to bind a thread to a processor, or to a virtual processor which in turn is the source of a processor binding to ultimately a processor.
* *CodeBinding* to bind source code associated with processes or threads to memory components.
* *DataBinding* to bind data components as well as source code data including stacks and heaps.
* *ConnectionBinding* to bind a connection to a flow sequence in hardware platform, or a virtual hardware platform whose elements are ultimately bound to a physical hardware platform.

Syntax

BindingType ::=

DataType

Legality Rules

1. A binding type must be a data type definition that is a map of a type union to another type union. The types in the type union must be data types that identify component categories (model elements) in an AADL model.

### Bindings

Description

1. A binding defines a deployment of a source component to a target component in terms of the specified binding type.
2. The source and target component can be several levels down the component hierarchy. However, a binding cannot reach into a subcomponent with a parameterized configuration. Instead the binding can identify a binding point in such a subcomponent.
3. The source may require resources and the target may provide resources. Such resource requirements and provisions may be represented user defined binding points, which are referenced by the binding definition.

Syntax

Binding ::=

Identifier **:** [ BindingTypeReference ] **binding**

ModelElementReference **->** ModelElementReference

[ InModes ] [ **{** { PropertyAssociation }+ **}** ]**;**

Legality Rules

1. The model element reference must resolve to a subcomponent in the component hierarchy or to a user defined binding point within a subcomponent. The model element reference must resolve to a subcomponent contained in a parameterized configuration.
2. If the binding type reference is present the source and target component must be one of the types specified in the binding type map.
3. If the binding type reference is present and the model element reference resolve to a binding point the binding type of the binding point must be the same as that of the binding definition.
4. If both model element references resolve to a binding point the binding type of both must be the same.

### Binding Points

Description

1. Binding points are named elements defined in the interface of a component and can be the source or target of a binding. This is useful when a component represents multiple resources, each available through a separate binding. It is also useful when a subcomponent is defined by a parameterized configuration, which limits visibility to its internal components.
2. Provides indicates that the binding point can be the target of a binding.
3. Requires indicates that the binding point can be the source of a binding.
4. Binding points can have properties that may indicate required or provided qualitative or quantitative resources.

Syntax

BindingPoint ::=

Identifier **:** ( **requires**  |  **provides** ) **binding** BindingTypeReference

[ **{** { PropertyAssociation }+ **}** ] **;**

Legality Rules

1. The component containing the binding point definition must be consistent with the component categories specified by the map of the binding type.

### Qualified and Quantified Resources

1. The source of a binding may require resources, while the target of a binding may provide those resources. For example, a thread may require processing cycles, while a processor provides processing cycles. The resource characteristics are represented by properties indicating the resource required by the binding source and provided by the binding target.
2. A processor may provide *Processing Cycles* as a quantified resource expressed by numeric values and an *Instruction Set* as a qualified resource expressed as categorical values through enumeration literals. Processing cycles is a property whose type is numeric with a unit, while instruction set is a property whose type is an enumeration identifying different instruction sets.

# Component Flows

## Flow Specifications

Description

1. A *flow specification* defines required flows across a component’s interface features providing a simplified abstraction of flows present within a component. This enables flow-related analyses of systems early in the development process where components do not have their implementation elaborated yet. It also enables compositional analysis one layer of the component hierarchy at a time. Flow specifications can be used to obtain better automation for the kinds of analysis commonly associated with naming conventions and/or feature groups. Flow specifications are more relevant for components whose interface allows for more than one kind of interaction with, or flow through that component.
2. Identifying a feature as a *flow sink* indicates that a flow terminates within a component, i.e., providing a required input to a specific flow analysis. Identifying a feature as a flow source indicates that a flow originates within a component, i.e., is required as an output. A flow path is an abstraction specifying that a flow from one or more incoming feature to one or more outgoing feature. The incoming and outgoing feature do not have to have the same feature category or type reference. Flows are elaborated in implementations as flow sequences through subcomponents.
3. The same incoming feature may be part of more than one flow specification. This may be a flow and a flow sink indicating that part of the flow may be filtered to end within the component, or multiple flows indicating a fan out.
4. Similarly, the same outgoing feature may be part of more than one flow specification. This may be a flow source and a flow indicating the component is both the source of an output as well as processes input to produce output.
5. Behavior specifications can be used to specify conditional flows such as processing two inputs to produce output. In its most general form such a specification consists of precondition and post condition specifications.

Syntax

FlowSpec ::=

Identifier **:**

( **flow source** ModelElementReferences

| **flow sink** ModelElementReferences

| **flow** ModelElementReferences **->** ModelElementReferences

[ InModes ] [ **{** { PropertyAssociation }+ **}** ]**;**

ModelElementReferences ::=

ModelElementReference | **(** ModelElementReference { **,** ModelElementReference }\* **)**

Legality Rules

1. The model element reference must resolve to a feature, possibly nested within one or more named interfaces, of the classifier containing the flow specification.
2. The model element reference of a flow source must resolve to an outgoing feature.
3. The model element reference of a flow sink must resolve to an incoming feature.
4. The first model element reference of a flow path must resolve to an incoming feature and the second model element reference to an outgoing feature.

## Flow Paths and Flow Sequences

Description

1. A flow path definition identifies two subcomponents and their flow specification as end points of a flow that is elaborated by a flow sequence.
2. A *flow sequence* represents a flow across multiple components. It consists of a sequence of flow steps that start and ends with a subcomponent with its flow specification. The sequence alternates between connections and subcomponents with their flow specification.
3. Flow sequences are used in two ways:

* As *flow sequence assignment* to a flow specification to elaborate a flow within a component implementation. In this case the end point(s) of the flow sequence must be connected to the features identified by the flow specification.
* As *flow sequence assignment* to a flow path definition to elaborate a flow between two subcomponents within an implementation. In this case the end points of the flow sequence must be the same as those identified by the flow path definition.

1. A flow sequence may reference a flow path instead of a subcomponent and flow specification allowing users to compose flow sequences from other flow sequences.
2. Flow paths and flow specifications of the top level component are instantiated by recursively elaborating their assigned flow sequences.
3. Optional: Users can specify subcomponent references without identifying a flow specification. In that case the features involved in the flow are identified by the preceding and succeeding connections.
4. Optional: If users specify subcomponent references with flow specifications, then the connections between those subcomponents can be inferred, i.e., the connection reference becomes optional.
5. Optional: Users may specify a flow sequence by skipping the subcomponent reference as it can be inferred from two successive connections.

Syntax

FlowSequenceAssignment ::=

FlowReference **=> flow** FlowSequence **;**

FlowPath ::=

Identifier : **flow** ModelElementReference **->** ModelElementReference

[ **{** { PropertyAssociation }+ **}** ] [ InModes ] **;**

FlowSequence ::=

SubflowReference { **->** ConnectionReference **->** SubflowReference }+

Legality Rules

1. A *ModelElementReference* in a flow path definition must resolve to a subcomponent or to flow specification in a subcomponent. The referenced subcomponent may be a nested subcomponent.
2. A *Subflow Reference* is a *Model Element Reference* that must resolve to a subcomponent, to flow specification in a subcomponent, or to a flow path. The referenced subcomponent may be a nested subcomponent.
3. A *Connection Reference* is a *Model Element Reference* that must resolve to a connection.
4. The source of a referenced connection must be the same as the outgoing feature of the preceding subcomponent flow specification.
5. The destination of a referenced connection must be the same as the incoming feature of the succeeding subcomponent flow specification.
6. In the case of a flow sequence assignment to a flow specification the incoming feature of a flow of flow sink specification must be the source of a connection delegation whose destination is the incoming feature of the first subcomponent flow specification reference.
7. In the case of a flow sequence assignment the outgoing feature of a flow of flow source specification must be the destination of a connection delegation whose source is the outgoing feature of the last subcomponent flow specification reference.
8. In the case of a flow sequence assignment to a flow path definition the end points of both must be the same.

Examples

**process** **interface** control **is**

insignal: **in** **port**;

outaction: **out** **port**;

// Flow path

processflow: **flow** insignal -> outaction;

**end**;

**process** control.impl **is**

dofilter: **thread** filter;

docompute: **thread** compute;

extin: **connection** insignal -> dofilter.insignal;

ftoc: **connection** dofilter.outsignal -> docompute.insignal;

extout: **connection** docompute.outsignal -> outaction ;

// Flow sequence assignment to elaborate a flow spec

processflow => **flow** dofilter.filterpath -> ftoc -> docompute.computepath ;

**end**;

**system** **interface** ControlSystem **is**

**end**;

**system** ControlSystem.i **is**

sense: **abstract** sensor.i;

processing: **process** control.impl;

actuate: **abstract** actuator.i;

hw : **system** hardwareplatform.impl;

sensetocontrol: **connection** sense.outp -> processing.insignal;

controltoactuate: **connection** processing.outaction -> actuate.inp;

etef: **end** **to** **end** **flow** sense.reading -> sensetocontrol-> processing.processflow -> controltoactuate -> actuate.action;

**end**;

# State Behavior

Description

1. State behavior is specified through state machines that are expressed by state variables and state transitions. State is affected by events, data, and typed tokens communicated through features and by component internal generators of events, data, and typed tokens.

## State Variables

Description

1. A *state variable* represents the current state of a given set of states. The set of states is represented by a set of literals in an enumeration type.
2. The initial state can be specified explicitly. Otherwise the first literal of the enumeration type represents the initial state.
3. A state variable is referenced in declarations that are conditional on a state or sets of states, e.g., when specifying architecture behavior based on modes or error behavior based on failure modes.
4. State variable can be declared for any component.
5. Behavior of the state machine associated with a state variable is expressed by state transitions.

Syntax

StateVariable ::=

Identifier **:** **state** EnumerationTypeReference [ **=** *Initial*StateReference ]

[ **{** { PropertyAssociation }+ **}** ] **;**

Naming Rules

1. The enumeration type reference must resolve to an enumeration type within the name space of the component.
2. The initial state reference must resolve to an enumeration literal within the enumeration type.

## State Transitions

Description

1. A *state transition* represents changes to state variables from their current state to a new state when a transition condition holds.
2. A state transition is executed if the state condition and the transition condition hold. The state condition allows state transitions to be defined from multiple source states to the same target state.
3. A transition condition can be based on presence of events and data on incoming features or conditions on their values. Transition conditions can also be based on typed tokens from incoming features or from generators.
4. If the optional state condition is missing then the transition is triggered from any current state if the transition condition holds.

Syntax

StateTransition ::=

Identifier **:** TransitionCondition **->** TargetStateReference [ **if** StateCondition ]

[ **{** { PropertyAssociation }+ **}** ] **;**

StateCondition ::=

StateVariableReference **in (** StateReference { **,** StateReference }\* **)**

TransitionCondition ::=

ConditionExpression

NOTE : The condition is expressed in the AADL V3 expression language.

Naming Rules

1. The state references must be one of the literals of the enumeration type associated with the state variable identified in the state condition.
2. The state references must resolve to an enumeration literal within the enumeration type of the state variable.
3. The condition expression must reference incoming features or binding points, data components, state variables, generators.

Examples

**package** StateTypes **is**

**type** threestate : **enum** ( s0 s1 s2);

**end**;

**system** gps.i **is**

mode : **state** threestate;

t1 : **transition** triggerevent -> s1 **if** s0 ;

**end**;

## Generators

Description

1. A *generator* represents the source of events, data values, or typed tokens within a component. This can be events like button pushes when no type is specified, data values like sensor readings or specific keys on a keyboard as indicated by the data type reference, or generated tokens of one of the specified types. For details on typed token see Section 13.1.
2. Properties specify the rate at which the generator produces events, data, or typed tokens.

Syntax

Generator ::=

Identifier **: generator** [ *Data*TypeReference | **{** TypeReference { **,** TypeReference }\* **}** ]

[ InModes ] [ **{** { PropertyAssociation }+ **}** ]**;**

# Modes and Mode-Specific Architecture Topology

Description

1. Architecture behavior represents dynamic changes to the topology of an architecture based on modes.
2. A mode defines a set of operational mode states and transitions. At any time only one mode is the current mode. Subcomponents and connections can optionally be defined to be active in a given set of mode. Component can have properties with different values for different modes.
3. Modes may optionally be defined to add realism to an AADL model, and modes allow analysis to focus on subsets and/or mode transitions. Modes can be defined differently for different components in a system. One mode is defined to be the initial mode.
4. A state variable represents a *mode*, which reflects dynamic behavior of an architecture in terms of its topology. In different modes different subsets of subcomponents and connections can be active. The architecture can be a platform architecture with hardware and physical components, or a task and communication architecture representing embedded software.
5. Subcomponents and connections can be declared as being active when the enclosing component is in a particular (set of) modes. This is expressed by a *state condition* that is associated with a subcomponent or connection declaration. The condition holds if the current value of the state variable is contained in the set of states specified by the condition.
6. Multiple components can have modes and each can change its current mode independently.
7. Dynamic changes to the architecture topology are expressed by state transitions. State transitions of a component state variable may be triggered by external input, subcomponent output, or by generators within a component.
8. A mode transition, i.e., a change to the topology of an architecture, may occur immediately, e.g., a failure of a hardware component, or it may be performed in a delayed fashion, e.g., activating and deactivating periodic threads at a hyper period. Further details of the execution semantics of mode transitions can be found in Part X.
9. Within a thread, state variables represent behavioral states whose behavior is elaborated in subprogram behavior rules without changes to the platform or task and communication architecture of an embedded system.

Syntax

InModes ::= **if** StateCondition

Examples

**system** gps.i **is**

mode : **state** threestate;

t1 : **transition** triggerevent -> s1 **if** s0 ;

locator : **process** sub.i1 **if** mode **in** ( s0 , s1 ) ;

**end**;

# Component Behavior

Description

1. Component behavior specification represents an abstraction of how a component processes input to produce output, possibly maintaining state. Component behavior specifications are an adaptation of the Behavior Annex to use the AADL V3 expression language.
2. A *behavior specification* of a component defines the computational behavior of a component in terms of processing its input to produce output and optionally maintaining state.
3. The behavior specification syntax will be an adaptation of the current Behavior Annex to utilize the AADL V3 expression language.

Syntax

Behavior ::=

Identifier **: behavior** Expression | ExpressionBlock **;**

ExpressionBlock ::= **{** Expression **}**

# Typed Token Behavior

Description

1. *Typed token behavior* specifications are an adaptation and extension of the error Model V2 Annex to accommodate fault behavior modeling as well as other behavior expressible by typed token system semantics.
2. A *typed token behavior* specification specifies that a component is the source of outgoing typed tokens, the sink of incoming typed tokens, and the component may propagate incoming typed tokens to other component via outgoing features and binding points. The component may possibly transform the incoming token(s) into a token of a different type.
3. One or more *Typed tokens* propagate from a typed token generator as source throughout the system following connections and bindings. Incoming typed token may be propagated as is through outgoing features and binding points or they may be transformed into a token of different type that is propagated to other components.
4. A typed token may represent Meta information associated with events or data communicated through features such as ports, or it may represent Meta information that propagates through the system separate from computational behavior, e.g., propagation of a service omission error type to indicate the lack of communication between two components.

Syntax

TokenBehavior ::=

Identifier **:**

( **token if** TokenCondition **then** TokenResultBlock **;**

| **token if** TokenCondition **then sink** [ **{** { PropertyAssociation }+ **}** ] **;**

TokenCondition ::=

MultiOperandExpression | TokenConditionElement

TokenConditionElement ::=

ModelElementReference [ **in (** TypeReference { **,** TypeReference }\* **)** ]

MultiOperandExpression ::=

LogialOperator **(** TokenConditionElement { **,** TokenConditionElement }\* **)**

LogicalOperator ::= **all**  | **any** | **one of** | k **of** | k **ormore** | k **orless**

TokenConditionElement ::=

StateCondition |

OriginatingModelElementReference [ **in (** TypeReference { **,** TypeReference }\* **)** ]

TokenResultBlock ::= **{** { TokenOutput | DetectionEvent | PropertyAssociation }+ **}**

TokenOutput ::=

OutgoingModelElementReference [ **=>** TypeReference ] **;**

DetectionEvent ::= FeatureReference **!** [ **(** Value **)** ]

NOTE: The token condition and token output declarations are a subset of the AADL V3 expression language. The detection event follows the syntax of the current Behavior Annex.

Legality Rules

1. The originating model element reference must be to a generator, an incoming feature or binding point.
2. The outgoing model element reference must be to an outgoing feature or binding point.

## Simple Token Propagation

Description

1. In its simplest form a typed token system specification consists of generator declarations for different typed tokens in various components and optionally token sink specifications in specific components.
2. Given such a specification instances of typed token propagate from generators through outgoing component features and binding points. The resulting propagation may consist of one or more typed tokens.
3. The typed tokens propagate following connections and bindings. The receiving components pass on incoming typed token through all outgoing features and binding points, or they act as sinks for tokens of specified types.
4. An analysis can determine whether any and which typed tokens reach a target component or an outgoing feature of the top-level system.

Examples

**interface** sensor **is**

outp: **out** **feature**;

fea: **feature**;

@SEC{

e1: **generator** (Virus, DirtyWord, Classified) ;

};

**end**;

**thread** **interface** compute **is**

insignal : **in** **feature**;

outsignal : **out** **feature**;

@SEC{

filter2: **token** insignal **in** (Virus,Classified) -> **sink**;

};

**end**;

## Token Propagation and Transformation

Description

1. A typed token system may include typed token behavior specification.
2. The condition may identify one or more incoming features and binding points without a type constraint or specify that incoming typed tokens must be of one of the specified types in order for the token behavior to be interpreted. If no type constraint is specified then any incoming typed token is considered as satisfying the condition element.
3. The token output specification may identify one or more outgoing features or binding points and optionally assign a token type. When no token type is assigned the collection of incoming typed tokens identified in the condition are passed on as outgoing tokens. If a token type is specified the collection of incoming tokens is transformed into a single token of the specified type.
4. In a typed token behavior specification, the condition may include or consist of one of more references to typed token generators. In this case the generated tokens are only pass on through the outgoing features and binding point specified as token output. A collection of generated tokens may even be transformed into a single outgoing token of a specified type.

Examples

**system** gps.i **is**

mode : **state** threestate;

## Stateful Typed Token Propagation

Description

1. Conditions on state variables may be included in the specification of token conditions. This allows users to specify token propagation and transformation behavior of components that is state sensitive, e.g., failure mode specific handling of incoming error types.
2. State transitions in a typed token system are triggered by typed tokens as specified by the transition condition, which is expressed as a token condition.

## Annotated Type Token Systems

Description

1. Typed token system specifications can be organized with annotations.
2. Data types, state variables, generators, and token behavior specifications may be declared with one of more associated annotation. Specifications with a specific annotation name (tag) are restricted to refers to data types with the same tag.
3. Analyses on typed token system specifications are performed with respect to specified annotations.
4. If a typed token system specification does not include annotations, then all data type references are acceptable in type references.

Examples

**interface** sensor **is**

outp: **out** **feature**;

fea: **feature**;

@SEC e1: **generator** (Virus, DirtyWord, Classified) ;

**end**;

**thread** **interface** compute **is**

insignal : **in** **feature**;

outsignal : **out** **feature**;

@SEC{

filter2: **token** insignal **in** (Virus,Classified) -> **sink**;

};

**end**;

## Error Behavior Specification

Description

1. The Error Model V2 Annex (EMV2) provided three levels of abstraction for error behavior specification (component error propagation, component error behavior, and composite error behavior) as well as taxonomy of error types.
2. In AADL V3 we use the annotation *@EM* to tag data type, state variables and transitions, generator, and token behavior declarations to indicate that they are part of an error behavior specification.
3. Error types are represented by data type definitions tagged with *@EM*. A in EMV2 error types can be organized into type hierarchies. As in EMV2 users can declare named type sets that can be referenced wherever collections of type are specified, e.g., in declarations of generators.
4. Error state machines are represented by state variable and state transition declarations tagged with *@em*. A component is limited to a single state variable representing an error behavior state machine.
5. Error propagation specifications are represented by token propagation and transformation specifications. In EMV2 these specifications are agnostic of error behavior states (no state condition) and identified pass through or transformation of a single incoming feature or binding point to a single outgoing feature or binding point of a component.
6. We allow error propagation specifications that include from multiple tokenized incoming features or binding points as well as multiple tokenized outgoing features or binding points.
7. We also allow token behavior specifications to be stateful, i.e., include state conditions as part of a token condition specification. State conditions must reference *@EM* tagged state variables. They may also reference state variables that represent component modes. Resulting of mode sensitive error behavior specifications.
8. This leads to an integration and in place evolution of EMV2 error propagation and transformation behavior and EMV2 component error behavior.
9. EMV2 component behavior specifications are represented as follows. EMV2 state transitions are expressed by state transitions tagged with *@EM*. Outgoing propagation conditions are represented by stateful typed token behavior specifications tagged with *@EM*. EMV2 error detection conditions are represented by stateful typed token behavior specifications tagged with *@EM*, that consists of a detection event as declaration in the token result block.
10. Finally, EMV2 composite error behavior specifications are represented by state transition declarations where the transition condition references state variables in subcomponents and the optional state condition is absent.

Examples

1. This example illustrate error behavior specification on a simple control system with replicate sensors.

**thread** **interface** sense **is**

p1: **out** **feature** ;

@EM{

s : **state** threestate;

esens1: **generator** (ServiceOmission) ;

sense1: **transition** esens1 **in** (ServiceOmission) -> s = s1 **if** s = s0 ;

};

**end**;

**thread** **interface** actuate **is**

p1: **in** **feature** ;

effect: **out** **feature** ;

@EM{

s : **state** threestate;

eact1: **generator** (ServiceOmission) ;

actuate1: **transition** eact1 **in** (ServiceOmission) -> s = s1 **if** s = s0 ;

};

**end**;

**thread** **interface** filter **is**

insignal1 : **in** **feature**;

insignal2 : **in** **feature**;

outsignal : **out** **feature**;

#Period => 20 ;

@EM{

filter2: **token** **all** (insignal1 **in** (ServiceOmission), insignal2 **in** (ServiceOmission)) ->

{ outsignal =ServiceOmission; }

};

**end**;

**thread** **interface** compute **is**

insignal : **in** **feature**;

outsignal : **out** **feature**;

@EM{

es: **state** threestate;

e1: **generator**;

r1: **transition** e1 -> es = s1 **when** es = s0;

compute2: **token** insignal **in** (ServiceOmission) -> outsignal=ServiceOmission;

r3: **token** insignal **in** (ValueError) -> **sink**;

};

**end**;

**system** **interface** conntop **is**

effect : **out** **feature**;

**end**;

**system** conntop.i **is**

sense1: **abstract** sensor.i;

sense2: **abstract** sensor.i;

processing: **process** controlProcess.impl;

actuate: **abstract** actuator.i;

hw : **system** hardwareplatform.impl;

sensetocontrol1: **connection** sense1.outp -> processing.insignal1;

sensetocontrol2: **connection** sense2.outp -> processing.insignal2;

controltoactuate: **connection** processing.outaction -> actuate.inp;

effectprop: **connection** actuate.effect -> effect;

@EM{

Effect1:**flow** actuate.actt2.effect **in** (ServiceOmission) -> effect=ServiceOmission;

};

**end**;

# Annotations

Description

1. An *annotation* allows users to tag an AADL model with information about the model rather than about the system being modeled. The latter is expressed by properties.
2. Annotations can be associated with any declaration, including package, classifier, data type, property definitions, as well as model element declarations such as subcomponents, features, flows, token behavior, and computational behavior.
3. One or more annotations can be associated with individual declarations and with collections of declarations.
4. An annotation has a name and optionally parameters.
5. Annotations are interpreted by model processing tools such as analysis tools. For example, an annotation named *em* is used to tag data type definitions to represent error types, state variable to identify error state machines, and token behavior declarations to represent a component or composite error behavior specification. Similarly, collections of data types may be tagged with multiple annotations to represent different subsets of token types and an analysis may allow the user to select a subset by the annotation name.

Syntax

Annotation ::=

**@** AnnotationIdentifer [ **(** { AnnotationParameter }\* **)** ]

AnnotationBlock ::= Annotation **{** Declarations **}**

AnnotationParameter ::= Name **=** Value

Legality Rules

1. The declarations in an annotation block must be of the same kind as those allowed in the context of the annotation block..

# Annex Subclauses and Annex Libraries

Description

1. An *annex subclauses* are expressed in a sublanguage to be attached to classifiers. Examples of such annexes are the Resolute, Agree, and BLESS.

NOTE: The Error Model V2 Annex and potentially the Behavior Annex are expressed in the AADL V3 core language as described in Sections 13.1 and 13.2

1. An *annex library* is a package that contains a collection of annex definitions expressed in one specific annex sublanguage. Those definitions can be referenced by annex subclauses of the same annex sublanguage.
2. A major use of these annex declarations is to accommodate new analysis methods through analysis-specific notations or sublanguages.
3. An annex subclause provides additional specification information about a component to be interpreted by analysis methods. Annex subclauses apply to component types and component implementations. Such annex subclauses can introduce analysis specific notations such as constraints and assertions expressed in predicate logic or behavioral descriptions expressed in temporal logic. Such notation can refer to subcomponents, connections, modes, and transitions as well as features and subcomponent access.

Syntax

EmbeddedAnnexSubclause ::=

**@** AnnexIdentifer **{\*** { AnnexSubclauseDeclaration }\* **\*}**

AnnexLibrary ::=

**package** PackageName **@** AnnexIdentifier **{\*** { AnnexDefinition }\* **\*}**

Naming Rules

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

Processing Requirements and Permissions

1. Processing methods compliant with the core AADL standard must accept AADL specifications with approved and project-specific annex subclauses and libraries, but are not required to process the content of annex subclauses and annex libraries. Processing methods compliant with a given annex sublanguage must process specifications as defined in that annex sublanguage.
2. Annex specific sublanguages can use any vocabulary word except for the symbol \*} representing the end of the annex subclause or library.
3. Annex specific sublanguages may introduce reserved words that may be the same or different from those in the core language or other annex sublanguages. If the annex sublanguage uses a reserved word that is a legal identifier in the AADL core language, then it must support the ability to refer to this named element in the core model.
4. Annex specific sublanguages can utilize the core language property mechanism, i.e., properties can be defined in property sets that apply to elements in the sublanguage annex. For example, a property occurrence can be defined to apply to an error event in an error model.
5. Annex sublanguages may choose not to support inheritance of sublanguage declarations contained in annex libraries of ancestor component type or component implementation declarations by their extensions.

Examples