SAE INTERNATIONAL

AEROSPACE STANDARD

AS5506™ REV. D

Draft 2019-05

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.

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

TABLE OF CONTENTS

No table of figures entries found.

1. 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 ::=
  package PackageName is
        { ImportDeclaration | PackageElement | NestedPackage }*
  end ;

PackageName ::= Identifier { :: Identifier }*

PackageElement ::=
  [ 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

(N1) 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.

(N2) A package provides a name space for definitions contained in the package.

(N3) The package name in a package element reference or a wildcard package reference must resolve in the global name space.

(N4) Each package introduces a namespace for its package elements.

(N5) 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.

(N6) An import reference to a package element makes the referenced definition visible to the package containing the import declaration.

(N7) The alias identifier resides in the name space of the package containing the import declaration.

(N8) 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

(L1) A package element marked as *private* must only be referenced within the package it is defined in.

(L2) A reference qualified by package name does not have to be made visible through an import declaration.

Processing Requirements and Permissions

(8) 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.

(9) Top-level packages, i.e., packages that are not syntactically nested, may be stored in separate files.

(10) 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

(11) 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;
```

2. 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

(N1) A classifier introduces a local name space for model elements defined within the classifier.

(N2) 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.

(N3) A classifier reference is resolved according to the naming rules for package element references.

(N4) 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.

3. INTERFACES

Description

(1) An *interface* represents the external interface specification of a component. The component interface provides a contract for the component that users of the component can depend on. All interactions of this component with other components 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,
- *flow specifications* indicating flow sources, sinks, and paths from incoming to outgoing features,
- *mode specifications* indicating externally visible operational mode behavior,
- *annex subclauses* that specify additional characteristics of the component,
- *associations of property value*s to the component interface and its elements, and
- *configuration assignments* to associate types with features of interfaces being extended.

(4) Interface extension allows users to represent related systems and provide libraries of reusable interface definitions.

(5) 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.

(6) An interface extension of multiple interfaces combines all elements of the interfaces become part of the resulting composite interface.

(7) 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.

(8) 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.

(9) 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 { InterfaceElement }* 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 | FlowSpecification | ModeSpecification | AnnexSubclause
  | PropertyAssociation | ConfigurationAssignment
```

Naming Rules

(N1) Features, binding points, flow specifications, and mode specifications are model elements whose defining names reside in the local name space of the interface.

(N2) 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.

(N3) An interface reference is resolved according to the naming rules for package element references.

Legality Rules

(L3) 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.

(L4) The category of an interface must not be *data*.

(L5) There must only be one property association for the same property in any of the interfaces being extended.

(L6) Configuration assignments must only be declared in interface extensions.

(L7) There must only be one configuration 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;
```

4. IMPLEMENTATIONS

Description

(1) An *implementation* represents the realization of a component that satisfies the interface identified by the implementation name. All external interactions only occur through the features of the interface, i.e., the interface enforces connectivity to external components.

(2) 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,
- *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,
- *mode specifications* that represent alternative operational modes that may manifest themselves as alternate configurations of subcomponents, connections, flow sequences, and property values,
- *annex subclauses* that specify additional characteristics of the component,
- *associations of property values* specific to the implementation and its contained elements, and
- *configuration assignments* to assign classifiers or data types to configure subcomponents of extended implementations and features of extended interfaces.

(3) An implementation extension can add implementation elements to an existing implementation. Furthermore, implementation extensions can refine existing subcomponents and features through configuration assignments and property associations.

(4) 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.

(5) The component hierarchy of an actual system is modeled by implementations with subcomponents, whose component classifier may identify another implementation with subcomponents.

Syntax

```
Implementation ::=
   Category ImplementationName
   [ extends ImplementationReference ]
   is { ImplementationElement }* end;

ImplementationName ::= Identifier . Identifier
```

```
ImplementationReference ::=
   [ PackageName :: ] ImplementationName

ImplementationElement ::=
   Subcomponent | Connection | Binding | FlowSequence | ModeSpecification
   | AnnexSubclause | PropertyAssociation | ConfigurationAssignment
```

Naming Rules

(N1) 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.

(N2) Subcomponents, connections, flow sequences, and mode specifications are model elements whose defining names reside in the local name space of the implementation.

(N3) An implementation inherits the name space of its associated interface.

(N4) An implementation extension inherits the name space of the implementation being extended.

(6) An implementation reference is resolved according to the package element reference naming rules.

Legality Rules

(L1) The category of the implementation must be the same as the category of the interface it is associated with.

(L2) An implementation cannot be associated with an interface without a category.

(L3) The category of an implementation extension must be the same as the category of the implementation being extended.

(L4) Configuration assignments must only be declared in implementation extensions.

Processing Requirements and Permissions

(7) An implementation denotes a set of actual system components, existing or potential, that are compliant with the implementation definition as well as the associated interface. Actual components denoted by different implementations for the same interface differ in additional details such as internal structure or behaviors; these differences may be specified using properties or annex subclauses.

(8) In general, two actual components 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;

  abstract interface actuator is
```

```
    state: provides in data access systemstate;
    inp: in port;
    fea: feature;
  end;

  abstract interface sensor is
    state: requires in data access systemstate;
    outp: out port;
    fea: feature;
  end;

  abstract sensor.i is
    sub1: thread sense;
    upmap: connection sub1.p1 -> outp ;
    stateaccess: connection state -> sub1.state;
  end;

  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;
```

5. SUBCOMPONENTS

Description

(1) A *subcomponent* represents a component instance. Subcomponents are instantiated when the containing implementation is instantiated. If the subcomponent declaration references an implementation or configuration, its subcomponents are instantiated recursively.

(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 can be configured through configuration 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.

Syntax

```
Subcomponent ::=
  Identifier : Category ( TypedSubcomponent | NestedSubcomponent ) [ InModes ] ;

TypedSubcomponent ::=
   TypeReference [ { { PropertyAssociation [ ; ] }+ } ] [ ; ]

TypeReference ::= ClassifierReference | DataTypeReference

NestedSubcomponent ::=
  { { PropertyAssociation | Subcomponent | Feature | Connection }+ }

SubcomponentReference ::=  Identifier
```

Legality Rules

(L1) 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.

(L2) If the category of a subcomponent declaration is of category *data*, then it must reference a data type.

(L3) 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

(5) 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

(6) The example illustrates the use is nested subcomponents. The example in the previous section illustrates the use of classifier based subcomponent definitions.

```
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;
```

6. 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, i.e., subcomponents and connections. Users can also add model elements other than components and connections.

(2) A configuration consists of a collection of

- *configuration 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 components to platform components,
- *flow specifications* for components and *flow sequences* across components,
- *annex subclauses* that specify additional characteristics for model elements,
- *property associations* to components and other model elements.

(3) 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.

(4) 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.

(5) A configuration can be parameterized. In this case the component hierarchy represented by the subcomponents can only be configured through the parameters.

(6) A configuration can be defined to contain only generally applicable configuration 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 configuration assignment patterns of such a configuration.

Syntax

```
Configuration ::=
  configuration ConfigurationName [ ConfigurationParameters ]
  extends ClassifierReference  { , ClassifierReference }*
  is { ConfigurationElement }+
  end ;

ConfigurationName ::= Identifier [ . Identifier ]

ConfigurationReference ::=
   [ PackageName :: ] ConfigurationName
   [ ( ConfigurationActual { , ConfigurationActual }* ) ]

ConfigurationParameters ::=
  ( ConfigurationParameter { , ConfigurationParameter }* )

ConfigurationParameter ::= Identifier : TypeReference

ConfigurationElement ::=
  ConfigurationAssignment | PropertyAssociation | ModeAssignment
  | ConfigurationAssignmentPattern
```

Naming Rules

(N1) 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.

(N2) 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.

(N3) 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

(L1) A configuration reference must only include parameter actuals if the configuration is defined as parameterized.

(L2) If a configuration extends multiple classifiers then at most one classifier reference must be to an interface or to an implementation.

(L3) If a configuration extends multiple classifiers then the interface of these classifiers must be the same.

(L4) 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.

(L5) A configuration defined with a single identifier must only contain configuration assignment patterns.

(L6) 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.

7. CONFIGURATION ASSIGNMENTS

Description

(1) A *configuration assignment* assigns one or more configurations and at most one component implementation to a subcomponent 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 configuration 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 configuration assignments to components further down the hierarchy.

(4) Configuration assignments declared in implementation extensions are subcomponent refinements. In that case a configuration 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 configuration 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) Configuration 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 configuration 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 *configuration assignment pattern* allows users to perform a configuration assignment to all model elements whose classifier is the same or an extension of the classifier specified in the pattern.

Syntax

```
ConfigurationAssignment ::=
  ModelElementReference =>
  ( ( ConfigurationValue [ Subconfiguration ] [ ; ] )
  | Subconfiguration ) ;


ConfigurationValue ::=
  DataTypeReference | ImplementationReference | ConfigurationReference
```

```
  | ConfigurationParameterReference

ConfigurationParameterReference ::= Identifier

Subconfiguration ::= { { ConfigurationElement }+ }

ConfigurationAssignmentPattern ::=
  all(ClassifierReference) =>
  ( ( ConfigurationValue [ Subconfiguration ] [ ; ] )
  | Subconfiguration ) ;
```

Naming Rules

(N1) The model element reference of configuration assignments is resolved in the context of the classifier that contains the configuration 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.

(N2) In the case of configuration assignments declared in subconfigurations the model element reference is resolved with respect to the enclosing subcomponent being configured.

(N3) The configuration parameter reference must identify a configuration parameter of the configuration that contains the configuration assignment.

Legality Rules

(L1) The model element reference must identify a subcomponent or feature in the component hierarchy of the component classifier that contains the configuration assignment.

(L2) For a subcomponent model element reference of a configuration assignment whose target is a data component the assigned value must be a data type.

(L3) 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.

(L4) If the assigned configuration is parameterized then parameter actuals must be included.

(L5) If one of the assigned classifiers is a parameterized configuration, then additional configurations are limited to assigning missing property values, flows, and annex specifications.

(L6) 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 configuration assignments declared in implementation extensions.

(L7) 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.

(L8) 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 configuration assignments declared in configurations.

(L9) The interface of the classifier(s) being assigned must be the same as the interface of the subcomponent referenced in its definition or refinement.

(L10) 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.

(L11) 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.

8. FEATURES

Description

(1) A *feature* represents an interaction point of a component. All external interactions of a component 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) We have 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.

(4) The type reference of a feature is optional and can be configured through a configuration assignment.

Syntax

```
Feature ::=
  FeatureName : [ FlowDirection ]
  ( PortFeature | AccessFeature | AbstractFeature | NamedInterface )
  [ { { PropertyAssociation  }+ } ] ;

FeatureName ::= Identifier

FlowDirection ::= in  |  out  | in out
```

Naming Rules

(N1) A feature name is part of the local name space of a component interface or of a named interface feature.

Legality Rules

(L1) Each component category section specifies which feature categories are legal for that component.

8.1 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.

(3) Incoming ports may trigger a dispatch or mode transition if they have the keyword event.

Syntax

```
PortFeature ::= [ event ] port [ DataTypeReference ]
```

Legality Rules

(L1) The port feature must have a flow direction.

(L2) 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;
```

8.2 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;
```

8.3 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;
```

8.4 Subprogram Parameters

Description

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

Syntax

```
SubprogramParameterFeature ::= parameter [ DataTypeReference ]
```

Legality Rules

(L1) The subprogram parameter feature must have a flow direction.

8.5 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

(N1) The named interface inherits the name space of the referenced interface definition.

(N2) 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

(L1) The named interface must not have a flow direction.

(L2) 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;
```

9. 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.

(2) 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.

9.1 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.

Syntax

```
Connection ::=
Identifier : connection ModelElementReference (  ->  |  <->  ) ModelElementReference
  [  {  { PropertyAssociation }+  }  ]  [ InModes ]  ;
```

Naming Rules

(N1) 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

(L1) 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.

(L2) 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.

(L3) 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.

(L4) 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.

(L5) The type reference of both referenced features or referenced feature and subcomponent must be the same.

(L6) 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.

(L7) 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..

9.2 Flow Specifications

Description

(1) A *flow specification* defines an abstraction of flows 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.

(2) A flow source indicates that a flow originates within a component, while a flow sink indicated that a flow ends within a component. A flow path indicates a flow from an incoming feature to an outgoing feature. The incoming and outgoing feature do not have to have the same feature category or type reference.

(3) The same incoming feature may be part of more than one flow specification. This may be a flow path and a flow sink indicating that part of the flow may be filtered to end within the component, or multiple flow paths indicating a fan out of the flow.

(4) Similarly, the same outgoing feature may be part of more than one flow specification. This may be a flow source and a flow path 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 ModelElementReference
  | flow sink ModelElementReference
  | flow path ModelElementReference -> ModelElementReference
  [ { { PropertyAssociation }+ } ] [ InModes ] ;
```

Legality Rules

(L1) The model element reference must resolve to a feature, possibly nested within one or more named interfaces, of the classifier containing the flow specification.

(L2) The model element reference of a flow source must resolve to an outgoing feature.

(L3) The model element reference of a flow sink must resolve to an incoming feature.

(L4) 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.

9.3 Flow Sequences

Description

(1) 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.

(2) Flow sequences are used in two ways:

- As *flow sequence assignment* to a flow specification to represent its elaboration 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 *end to end flow* that starts and ends within the same component implementation. In this case the end points of the flow sequence represent the end points of the end to end flow.

(3) A flow sequence may reference an end to end flow instead of a subcomponent and flow specification allowing users to compose flow sequences from other flow sequences.

(4) End to end flows and assigned flow sequences of the top level component flow specifications are instantiated by recursively elaborating flow specifications with their assigned flow sequences.

(5) 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.

(6) 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.

(7) Optional: Users may specify a flow sequence by skipping the subcomponent reference as it can be inferred from two successive connections.

Syntax

```
FlowSequenceAssignment ::=
  FlowSpecificationReference => flow FlowSequence ;

EndToEndFlow ::=
  Identifier : end to end flow FlowSequence
    [ { { PropertyAssociation }+ } ] [ InModes ] ;

FlowSequence ::=
  SubflowReference { -> ConnectionReference -> SubflowReference }+
```

Legality Rules

(L1) A *Subflow Reference* is a *Model Element Reference* that must resolve to a flow specification in the referenced subcomponent or to an end to end flow. The referenced subcomponent may be a nested subcomponent.

(L2) A *Connection Reference* is a *Model Element Reference* that must resolve to a connection.

(L3) The source of a referenced connection must be the same as the outgoing feature of the preceding subcomponent flow specification.

(L4) The destination of a referenced connection must be the same as the incoming feature of the succeeding subcomponent flow specification.

(L5) In the case of a flow sequence assignment the incoming feature of a flow path of flow sink must be the source of a connection delegation whose destination is the incoming feature of the first subcomponent flow specification reference.

(L6) In the case of a flow sequence assignment the outgoing feature of a flow path of flow source must be the destination of a connection delegation whose source is the outgoing feature of the last subcomponent flow specification reference.

Examples

```
process interface control is
insignal: in port;
outaction: out port;
processflow: flow path 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 ;
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;
```

10. 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.

10.1 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

(L1) 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.

10.2 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
    [ { { PropertyAssociation }+ } ] [ InModes ] ;
```

Legality Rules

(L1) 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.

(L2) If the binding type reference is present the source and target component must be one of the types specified in the binding type map.

(L3) 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.

(L4) If both model element references resolve to a binding point the binding type of both must be the same.

10.3 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

(L1) The component containing the binding point definition must be consistent with the component categories specified by the map of the binding type.

10.4 Qualified and Quantified Resources

(1) Bindings may involve resources. 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 and an *Instruction Set* as a qualified resource. 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.

11. MODES AND MODE TRANSITIONS

Description

(1) A mode represents an operational mode state. Modes can be defined for different components in a system. One mode is defined to be the initial mode. At any time one mode is the current mode.

(2) Model elements can be specified to be active in a given set of modes of the enclosing component, if the current mode is one of the specified modes.

(3) Properties can have different values for different modes.

(4) A mode transition models dynamic operational behavior by transitioning the current mode to a different mode. This affects which model elements are active at any given time and which property values hold.

(5) Mode transitions are triggered by arrival of messages from external components, from subcomponents, or from error detection events. A mode transition may be performed immediately if it is considered an emergency, or it may be performed in a planned fashion by coordinating the set of components becoming active and inactive.

(6) For example, the current mode of a process determines the set of threads that are considered active. This allows users to define mode specific subsets of components, connections, and property values, i.e., change the topology of a system architecture dynamically. For example, users can model that different subsets of threads are actually executing during different operational modes, or that a hardware platform may have lost a component due to failure.

(7) Within a thread mode states represent behavioral states whose behavior is elaborated through the behavior annex.

(8) Multiple components can have modes and each can change its current mode independently.

Syntax

```
Mode ::=
    Identifier : [ initial ] mode
      [ { { PropertyAssociation }+ } ] [ ; ];

ModeTransition ::=
    Identifier : ModeReference -[ [ ModeTransitionTriggers ] ]-> ModeReference
      [ { { PropertyAssociation }+ } ] [ ; ];

ModeReference :== Identifier
```

```
ModeTransitionTrigger ::= ModelElementReference
```

```
InModes ::=
    in modes ( ModeReference [ ModeDelegation ]  { , ModeReference [ ModeDelegation ] }* )
```

```
ModeDelegation ::=
    => ModeReference
```

Naming Rules

(N1) The mode reference identifier must resolve to a mode defined in the component with the mode reference, except the mode reference of a mode delegation must resolve to a mode defined in the subcomponent with the *in mode*s specification.

(N2) The model element reference representing a mode transition trigger must resolve to a port of the enclosing component, a port of a subcomponent, or to an error detection event.

Legality Rules

(L1) For each component with mode state definitions, only one mode must be defined as *initial*.

(L2) A mode delegation must only be specified for *in modes* of a subcomponent.

Consistency Rules

(C1) The set of transitions declared within a single component implementation must define a deterministic transition function.

12. ANNEX SUBCLAUSES AND ANNEX LIBRARIES

Description

(1) An *annex subclauses* allow annotations expressed in a sublanguage to be attached to classifiers. Examples of standardized sublanguages are defined in the Error Model Annex and in the Behavior Model Annex.

(2) 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.

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

(4) 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.

(5) Discussion: Should we allow annex subclauses to be attached model elements inside a classifier? E.g., directly to a feature to represent the error type of a propagation?

Syntax

```
EmbeddedAnnexSubclause ::=
```

```
    @ AnnexIdenditifer  {* { AnnexSubclauseDeclaration }* *}

AnnexLibrary ::=
    package PackageName @ AnnexIdentifier  {* { AnnexDefinition }* *}
```

Naming Rules

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

Processing Requirements and Permissions

(6) 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.

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

(8) 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.

(9) 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.

(10) 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