Processing Requirements and Permissions

(147) A method of implementation is permitted to enforce that the with declaration in a package not be changed to enforce the use restrictions between packages when classifiers are added to the package.

Examples

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

4.3 Component Types

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

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

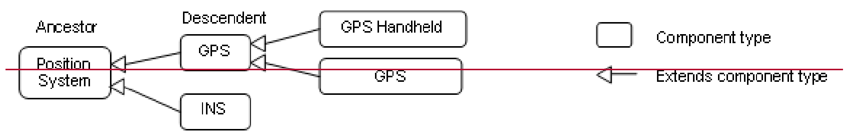
(150) Component types can declare modes and mode transitions of a component. These modes and mode transitions are common to all implementations of the component. This allows for mode-specific property values to be associated with the component type, its features, and its flows.

~~***(151)*** Component types can be declared in terms of other component types, i.e., a component type can *extend* another component type – inheriting its declarations and property associations. If a component type extends another component type, then features, flows, and property associations can be added to those already inherited. A component type extending another component type can also refine the declaration of inherited feature and flow declarations by more completely specifying partially declared component classifiers of features and by associating new values with properties. A component type cannot be an extension of multiple component types, i.e., multiple inheritance is not supported for AADL components~~

~~***(152)***~~ ***(151)*** Component types can declare *prototypes*, i.e., classifier parameters that are used in features. ~~The prototype bindings are supplied when the component types is being extended or used in subcomponent declarations~~The actual classifiers are supplied when the component type is used in subcomponent declarations or component type extensions.

~~***(153)*** Component type extensions form an *extension hierarchy*, i.e., a component type that extends another component type can also be extended. We use AADL graphical notation (see APPENDIX D) to illustrate the~~

~~extension hierarchy in Figure 12. For example, component type GPS extends component type Position System inheriting ports declared in Position System. It may add a port, refine the data type classifier of a port incompletely declared in Position System, and overwrite the value of one or more properties. Component types being extended are referred to as *ancestors*, while component types extending a component type are referred to as *descendents*.~~



~~Figure 12 – Component type extension hierarchy~~

~~*(154)*Component types may also be extended using an annex_subclause to specify additional characteristics of the type that are not defined in the core of the AADL (see Section (198))~~

Syntax

component_type ::=
  component_category *defining_component_type*_identifier
  [prototypes ({ prototype }$^+$ | none_statement)]
  [features ({ feature }$^+$ | none_statement)]
  [flows ({ flow_spec }$^+$ | none_statement)]
  [modes_subclause | requires_modes_subclause]
  [properties (
    { *component_type*_property_association | contained_property_association }$^+$
    | none_statement)]
  { annex_subclause }*
  end *defining_component_type*_identifier ;

~~component_type_extension ::=~~
~~component_category *defining_component_type*_identifier~~
~~extends unique_component_type_reference [prototype_bindings]~~
~~[prototypes ({ prototype | prototype_refinement }$^+$ | none_statement)]~~
~~[features ({ feature | feature_refinement }$^+$ | none_statement)]~~
~~[flows ({ flow_spec | flow_spec_refinement }$^+$ | none_statement)]~~
~~[modes_subclause | requires_modes_subclause]~~
~~[properties (~~
~~{ *component_type*_property_association | contained_property_association }$^+$~~
~~| none_statement)]~~
~~{ annex_subclause }*~~
~~end *defining_component_type*_identifier ;~~

```
component_category ::=
      abstract_component_category
    | software_category
    | execution_platform_category
    | composite_category

abstract_component_category ::= abstract

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

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

composite_category ::= system

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

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

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

Naming Rules

(N19) The defining identifier for a component type must be unique in the namespace of the package within which it is declared.

(N20) Each component type has a local namespace for defining identifiers of prototypes, features, modes, mode transitions, and flow specifications. That is, defining prototype, defining feature, defining modes and mode transitions, and defining flow specification identifiers must be unique in the component type namespace.

~~(N21) The component type identifier of the ancestor in a component type extension, i.e., that appears after the reserved word extends, must be defined in the specified package namespace. If no package name is specified, then the identifier must be defined in the namespace of the package the extension is declared in.~~

~~(N22) When a component type extends another component type, a component type namespace includes all the identifiers in the namespaces of its ancestors.~~

~~(N23) A component type that extends another component type does not include the identifiers of the implementations of its ancestors.~~

~~(N24)~~(N21)The defining identifier of a feature, flow specification, mode, mode transition, or prototype must be unique in the namespace of the component type.

~~(N25) The refinement identifier of a feature, flow specification, or prototype refinement refers to the closest refinement or the defining declaration of the feature going up the component type ancestor hierarchy.~~

~~(N26) The prototypes referenced by prototype binding declarations must exist in the local namespace of the component type being extended.~~

~~(N27)~~(N22)Mode transitions declared in the component type may not refer to event or event data ports of subcomponents.

Legality Rules

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

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

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

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

~~(L5)~~(L3)A component type must not contain both a requires_modes_subclause and a modes_subclause.

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

Standard Properties

Classifier_Substitution_Rule: inherit enumeration (Classifier_Match, Type_Extension, Signature_Match)

Prototype_Substitution_Rule: inherit enumeration (Classifier_Match, Type_Extension, Signature_Match)

Semantics

~~(155)~~ (152)A component type represents the interface specification of a component, i.e., the component category, prototypes, features, flow specifications, modes, mode transitions, and property values of a component. A component implementation of this component type denotes a component, existing or potential, that is compliant with the component type declaration. Component implementations are expected to satisfy these externally visible characteristics of a component. The component type provides a contract for the component interface that users of the component can depend on.

~~(156)~~ (153)The component categories are: data, subprogram, subprogram group, thread, thread group, and process (software categories); processor, virtual processor, bus, virtual bus, memory, and device (execution platform categories); system (compositional category), and abstract component (compositional category). The semantics of each category will be described in sections 5, 6, and 7.

~~(157)~~ (154)Features of a component are interaction points with other components, i.e., ports and feature groups; subprogram parameters; data component access, subprogram access, and bus access. Ports represent directional flow of data and events between components, feature groups represent groups of features that are connected to another component, data component access represents access to shared data components, subprogram access represents access to a subprogram by a caller, and bus access represents access to a bus from processor, memory, device, and other bus components to establish hardware connectivity. Features are further described in Section 8.

~~(158)~~ (155) Flow specifications indicate whether a flow of data or control originates within a component, terminates within a component, or flows through a component from one of its incoming ports to one of its outgoing ports.

~~(159)~~ (156) Mode declarations define modes of the component that are common to all implementations. As a result, component types can have mode-specific property values. Other components can initiate mode transitions by supplying events to incoming event ports of a component. Mode transitions specify which event ports affect their transition. A component type extension can add modes and associate properties to existing modes.

~~(160)~~ (157) A requires_modes_subclause specifies as set of modes that the component expects to inherit from its containing component. In this case, the component can utilize these modes for mode-specific property values and for in modes declarations of subcomponents and connection, but mode transition behavior is determined by the containing component, whose modes are made accessible to the component.

~~(161)~~ (158) A component type can contain *incomplete* feature declarations, i.e., declarations with no *component classifier references* or just the component type name for a component type with more than one component implementation. The component implementation may not exist yet or one of several implementations may have not been selected yet.

~~(162)~~ (159) A component type may also be declared with *prototypes*, indicating that the component type is incomplete and can be parameterized. The use of incomplete declarations is particularly useful during early design stages where details may not be known or decided. Classifiers ~~and features~~ can be supplied through prototype bindings ~~to complete such a component type template~~ as part of a subcomponent declaration, component type extension, component implementation extension~~, or when used as a component type reference, e.g., in feature declarations or subcomponent declarations~~. Once bound a prototype cannot be rebound. ~~The use of incomplete declarations is particularly useful during early design stages where details may not be known or decided.~~

~~(163) A component type can be declared as an extension of another component type resulting in a component type extension hierarchy, as illustrated in Figure 12. A component type extension can refine an abstract component type to one of the concrete component categories. A component type extension inherits the features, flow specifications, modes, mode transitions, prototypes, and properties of the component type being extended. For annex subclauses, each annex defines whether annex declarations are inherited. A component type extension can contain refinement declarations to permit incomplete feature declarations to be completed and new property values to be associated with features and flow specification declared in a component type being extended. In addition, a type extension can add feature declarations, flow specifications, modes, mode transitions, and property associations.~~

~~(164) A component type being extended may include prototype binding declarations. If prototype bindings are declared for a subset of the prototypes, then only the prototypes without binding can be bound when referencing the component type extension. This supports evolutionary development and modeling of system families by declaring partially complete component types that get refined in extensions.~~

~~(165)~~ (160) Properties are predefined for each of the component categories and will be described in the appropriate sections. See Section 11.3 regarding rules for determining property values.

Examples

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

```
end File_System;

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

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

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

4.4 Component Type Composition and Extension

(161) Component types can be declared in terms of other component types, inheriting features, flows, modes, annex subclauses, and property associations. A component type can *combine* multiple existing component types; and a component type can *extend* an existing component type by adding to or refining inherited declarations.

(162) The component type elements being inherited are either included in the name space of the extending component type, or their name space is identified by a user defined identifier. The latter allows users to include multiple instances of a component type and to resolve name conflicts between different inherited namespace.

(163) Component type compositions and extensions form a hierarchy. We use AADL graphical notation (see APPENDIX D) to illustrate the extension hierarchy in Figure 12. For example, component type GPS extends component type Position System inheriting ports declared in Position System. It may add a port, refine the data type classifier of a port incompletely declared in Position System, and overwrite the value of one or more properties. Component types being extended are referred to as *ancestors*, while component types extending a component type are referred to as *descendents*.

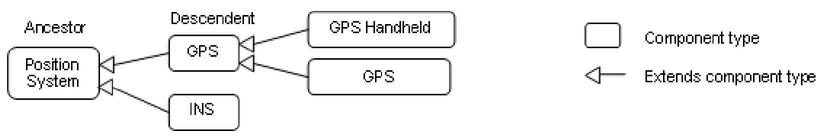


Figure 12 – Component type extension hierarchy

Syntax

component_type_composition_and_extension ::=
component_category *defining_component_type*_identifier
extends component_interface { **,** component_interface }*
[local_component_type_extension]
end *defining_component_type*_identifier ;

component_interface ::=
[*defining_interface*_identifier **:**] [**inverse of**] unique_component_type_reference [prototype_bindings]

local_component_type_extension ::=
[**prototypes** ({ prototype | prototype_refinement }$^+$ | none_statement)]
[**features** ({ feature | feature_refinement }$^+$ | none_statement)]
[**flows** ({ flow_spec | flow_spec_refinement }$^+$ | none_statement)]
[**modes** subclause | requires_modes_subclause]
[**properties** (
{ *component_type*_property_association | contained_property_association }$^+$
| none_statement)]
{ annex_subclause }*

Naming Rules

(N23) The defining identifier for a component type extension must be unique in the namespace of the package within which it is declared.

(N24) The defining interface identifier of a component being extended must be unique in the namespace of the component type extension.

(N25) The component type identifier of an ancestor in a component type extension, i.e., that appears after the reserved word *extends*, must be defined in the specified package namespace. If no package name is specified, then the identifier must be defined in the namespace of the package the extension is declared in.

(N26) When a component type extends another component type, its namespace includes all the identifiers in the namespaces of its ancestors, if no *defining interface identifier* is specified. If a *defining interface identifier* is specified references to elements of the inherited component type are qualified by this identifier.

Formatted: Font: Not Bold
Formatted: Font: Bold
Formatted: Font: Italic
Formatted: Font: Bold
Formatted: Font: Italic
Formatted: Font: Italic

(N27) The refinement identifier of a feature, flow specification, or prototype refinement refers to the closest refinement or the defining declaration of the feature going up the component type ancestor hierarchy.

(N28) The prototypes referenced by prototype binding declarations must exist in the local namespace of the component type being extended.

Legality Rules

(L4)The category of the component type being extended must match the category of the extending component type, i.e., they must be identical or the category being extended must be abstract.

(L5)If the extended component type and an ancestor component type in the extends hierarchy contain modes subclauses, they both must either be requires_modes_subclause or modes_subclause.

Standard Properties

Classifier_Substitution_Rule: inherit enumeration (Classifier_Match, Type_Extension, Signature_Match)

Prototype_Substitution_Rule: inherit enumeration (Classifier_Match, Type_Extension, Signature_Match)

Semantics

*(164)*A component type can be declared as a composition and extension of other component types resulting in a component type extension hierarchy, as illustrated in Figure 12. A component type composition and extension inherits the features, flow specifications, modes, mode transitions, prototypes, and properties of the component type being extended. These inherited elements are either included in the namespace of the component type extension when no defining interface identifier is declared, or their references must be qualified by the user defined *interface identifier* for the referenced component type. Qualification by defining interface identifier allows users to resolve name conflicts between different component type elements being composed, or to include multiple instances of the same interface, e.g., when a voter component receives the same input from two sources.

*(165)*Qualifying interface identifiers allow users to reference the interface as a whole, e.g., when declaring connections. Component types may be composed of component types with defining interface identifiers, which themselves are composed. This leads to nested qualifying interface identifiers. Note that this allows users to represent the feature group concept found in AADL V2.

*(166)*For component types with directional features, the extension may inherit the features of a component types as features with opposite direction, as indicated by the reserved words **inverse of**. For example, a component type may define a logical interface. The features of this interface definition can then be included in the component types of both a sending and a receiving component.

*(167)*A component type extension can add new and refine inherited features, flow specifications, modes, property associations, and annex subclauses.

*(168)*For annex subclauses, each annex defines the rules for inheriting individual declarations inside an annex.

Examples

package CompositionExtensionExample

public

abstract Logical

temperature: **out data port**;

Speed: **out data port**;

end Logical;

Commented [PF1]: Refinement of features in named interface.

Formatted: Font: Bold

Formatted: Font: Bold

Formatted: Font: Arial

Formatted: Font: Arial

```
abstract Physical
Network: requires bus access CANBus;
end Physical;
-- local extension of inherited type
system s1 extends Logical
features
Onemore: out event port;
end s1;
-- interface composition of two component types
system s2 extends Logical, Physical
end s2;
-- interface composition of two component types and local addition
system s3 extends Logical, Physical
features
Onemore: out event port;
end s3;
-- interface composition of two instances of the same component type
system voter extends Lin1: Logical, Lin2: Logical, Lout: Logical
end s2;
end CompositionExtensionExample;
```

Formatted: Font: Arial

Formatted: Font: Arial

Formatted: Font: Not Bold

Formatted: Font: Arial

Formatted: Font: Arial

Formatted: Font: Arial

Formatted: Font: Arial

Formatted: Font: Arial

Formatted: Font: Arial

Formatted: Font: Arial

Formatted: Font: Arial

Formatted: Font: Bold

~~4.4~~4.5 Component Implementations

~~(166)~~ (169) A *component implementation* represents the realization of a component in terms of subcomponents, their connections, flow sequences, properties, component modes and mode transitions. Flow sequences represent implementations of flow specifications in the component type, or end-to-end flows to be analyzed. Modes represent alternative operational modes that may manifest themselves as alternate configurations of subcomponents, connections, call sequences, flow sequences, and property values.

~~(167)~~ (170) A component type can have zero, one, or multiple component implementations. If a component type has zero component implementations, then it is considered to be a leaf in the system component hierarchy. For example, a partial AADL model may have processes as components without realization, while a task level AADL model expand to threads as leaves. If no implementation is associated then the properties on the component types provide information about the component for analysis and system generation.

~~(168)~~ (171) A component implementation can be declared as an extension of another component implementation. In that case, the component implementation inherits the declarations of its ancestors as well as its component type. A component implementation extension can refine inherited declarations, and add subcomponents, connections, subprogram call sequences, flow sequences, mode declarations, and property associations.

~~(169)~~ (172) Component implementations can declare *prototypes*, i.e., classifier parameters that are used in subcomponent declarations. The prototype bindings are supplied when the component implementation is being extended or used in subcomponent declarations.

~~(170)~~ (173) Component implementations build on the component type *extension hierarchy* in two ways. First, a component implementation is a realization of a component type (shown as dashed arrows in Figure 13). As such it inherits features and property associations of its component type and any component type ancestor.