Annex C AADL Meta Model and Interchange Formats

Normative

Annex C.1 Scope

This annex defines the AADL meta model and XML-based interchange formats for AADL models. The AADL meta model defines the structure of AADL models, i.e., an object representation of AADL specifications that corresponds to a semantically decorated abstract syntax tree. The object representation of AADL models can be manipulated programmatically through an API. The object representation of AADL models can also be persistently stored as XML documents in a standard interchange format. This permits different tools that support the AADL XML schema [XML 2001] or XMI meta model specification [XMI 2003] to interoperate on AADL models. Both the XML schema and the XMI meta model specification for the AADL are derived from the AADL meta model, thus, the two representations are consistent with the meta model.

The three model representations and persistent XML formats of AADL models defined in this annex are:

- A *declarative AADL model*: AADL specifications in the form of component types, component implementations, port group types, annex libraries, packages, and property sets.
- An *AADL instance model*: A compact representation of a system instance whose root is a system implementation in an AADL specification. This model contains AADL properties required for further analysis and processing. Declarative information is accessible, if needed, through cross-document references to the declarative model.
- A *graphical layout model*: Layout information for the graphical display of AADL models. The graphical layout representation is associated with AADL models in their declarative model and instance model.

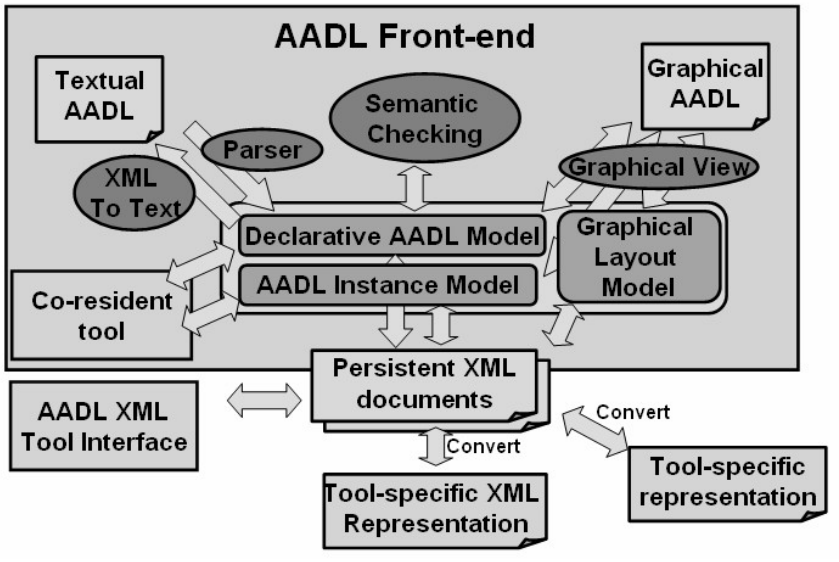

Figure 1 AADL Models and Tools

Figure 1 shows the three model representations and their roles with respect to the textual and graphical AADL representation as well as to tools that process and analyze AADL models.

The declarative AADL model reflects AADL specifications and supports translation to and from the textual AADL representation. A parser translates AADL text into a declarative AADL model. Name resolution, semantic checking and other static analysis can be performed on this in-core object model of the declarative AADL model. The emphasis of the declarative model representation is to maintain the declarative structure as defined in the standard. Derived information such as features inherited by subcomponents from their component type, subcomponents or connections inherited by a component implementation from the component implementation it extends, or property values of subcomponents inherited from property associations declared with the referenced component type or implementation are not explicitly modeled, but are accessible via methods that operate on the model. Results of such analysis can be recorded in the declarative AADL model as property values. This makes analysis results available in a standard format for other analysis and for generation tools as illustrated in Figure 1. An XML-to-Text converter can reproduce textual AADL from the declarative AADL model.

The graphical AADL model is supported by graphical editing tools operating on the declarative AADL model and using the graphical layout model to maintain relevant layout information about the AADL model. Graphical presentation of AADL instances is supported in a similar manner.

The AADL instance model reflects an instance of a system as an instantiation of the application system and the execution platform based on a system implementation as the root of the system model. Connections between connections, represented in the declarative model as one or more connection declarations, are resolved to semantic connections in the instance model. The declarative AADL model contains information that is relevant to assure the semantic consistency of AADL models. However, this information is often not needed for model analyses and runtime system generation. Therefore, this information is not kept in the instance model, but accessible through references from the AADL instance model objects to the model objects in the declarative model they are derived from. The AADL instance model is used in the analyses of system instances. In many cases these analyses require a system model in which application components are bound to execution platform components. Such analysis tools may operate on an memory-resident representation of the AADL instance model directly, or they may load an XML document into memory as part of the analysis process. Alternatively, filters can convert the XML representation into an analysis tool specific representation. Results from such analysis can be recorded as properties in the AADL instance model and mapped back into the declarative AADL model as necessary.

The three models of AADL specifications in this annex are described by the AADL meta model. This meta model is the basis of a standard persistent interchange format in XML as well as the basis of an in-core object model that results from loading an XML-based model into memory. The persistent XML representation is specified by an XMI meta model of the AADL and by an XML Schema. Both these specifications are directly derived from the AADL meta model.

This annex provides a mapping from the AADL meta model to XML documents as a persistent form of AADL models. The annex documents the decisions that went into the definition of the AADL meta model. And it provides a summary of the AADL meta model. The full AADL meta model and XML/XMI representations are available in an Appendix.

Annex C.2 Meta Model Notation

The AADL meta model was developed using the Eclipse Modeling Framework (EMF) [EMF 2003] Ecore notation, a meta modeling notation that is widely used elsewhere. EMF provides a meta modeling notation called Ecore. This tool environment also generates an XML schema and an XMI meta model from the AADL meta model defined in Ecore. Furthermore, the EMF tool environment generates XML document readers and writers that operate with the EMF meta model, the XMI meta model, or the XML schema.

The meta model is represented as a set of class diagrams with additional EMF-specific properties that support the automatic generation of methods for manipulation of AADL object models and for persistent storage and retrieval of such models in one or more XML documents.

Annex C.2.1The Graphical Meta Model Notation

A UML class diagram notation has been used to graphically present the Ecore meta model of AADL (see Figure 2). Classes are distinguished from other symbols by a little table-like icon in the upper left corner. A hierarchy of *abstract* and *concrete* classes is used to represent the different object classes of an AADL model. Concrete classes can be instantiated into AADL model objects. Abstract classes represent characteristics that are common to a group of AADL model objects. For example, the abstract class *NamedElement* represents AADL model objects that have a name. A line with a hollow triangle denotes a sub class super class relationship and points to the super class. Abstract classes are marked with the letter A in the upper left corner.

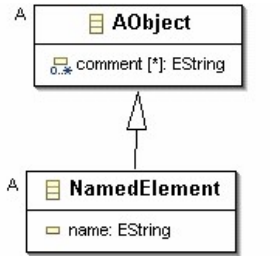

Figure 2 Meta Model Class Notation

Classes in the meta model can also contain attributes, i.e., typed name value pairs. The attribute values can be of a number of predefined types or can be enumeration literals of user-defined enumerations. Figure 2 shows a multi-valued comment attribute and a single-valued name attribute.

Objects in an AADL model are related to each other through two types of associations: a *containment association*, and a *reference association*. A containment association specifies that one object is part of another object. The UML symbol for composition aggregation, i.e., a line with a diamond at the container end and an open arrowhead at the contained end, is used to represent containment associations (see the containment of *ProcessSubcomponents* in *ProcessImpl* in Figure 3).

A reference association specifies that objects are accessible from other objects. Reference associations can be unidirectional, i.e., one object can access another object, or bi-directional, i.e., either object can access the other object. Unidirectional reference associations are shown as a line with an arrowhead and a label at the destination, the UML symbol for unidirectional associations (see the reference from *ProcessSubcomponent* to *ProcessClassifier* in Figure 3). Bi-directional reference associations are shown as a line without an arrowhead and labels on both ends, the UML symbol for bi-directional associations (see the reference between *ProcessType* and *ProcessImpl* in Figure 3).

For bi-directional reference associations, the reference to another object is kept consistent with the reference in the opposite direction. If one of the labels of a bi-directional reference association is marked with a (T), then references in this direction are transient, i.e., not persistently stored in XML. On loading an XML document, the reference marked as transient is re-established for the in-core representation.

Both containment and reference associations can have a multiplicity to specify the number of association instances. Figure 3 shows a containment association between *ProcessImpl* and *ProcessSubcomponents*. The association indicates that a process implementation can contain zero or one process subcomponents subclause objects. Figure 3 also shows a unidirectional reference association from a process subcomponent to a process classifier and indicates that the classifier reference is optional, i.e., zero or one. For the bi-directional reference association between process types and process implementations, the reference from the implementation to the type is required to be one and is stored persistently, while the reference from the type to the implementations is zero or more and is maintained only in-core.

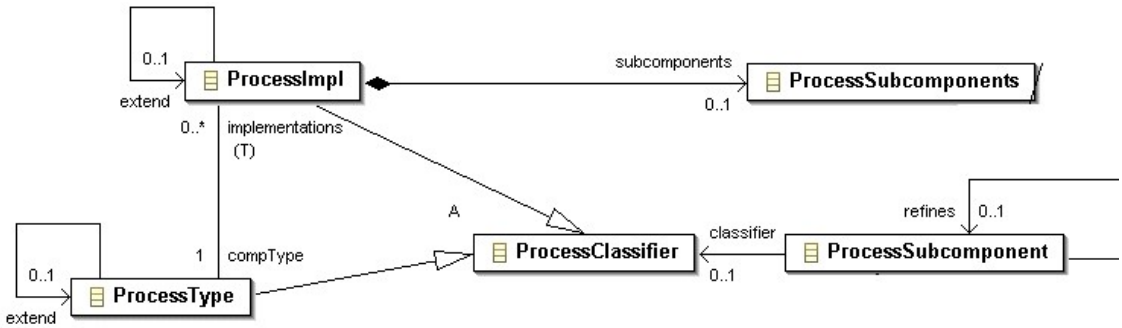

Figure 3 Containment and Reference Associations

Reference associations between classes may be marked with an S for *specialize* (see Figure 4) if one or both are abstract classes. Subclasses of these abstract classes specify specializations of this reference that restrict the endpoints of the reference to the objects of the classes involved in the specialized associations. For example, the reference association between *ProcessType* and *ProcessImpl* in Figure 3 limits the reference to implementations and types of processes.

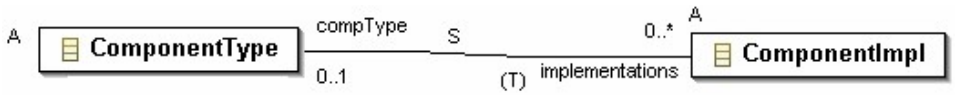

Figure 4 Relations with Specialization

Annex C.2.2Mapping to XML Documents

The focus of the AADL meta model is to represent the declarative structure of AADL models as an object model, with relevant cross-references directly expressed in the object model, and to map this object model into an XML document.

An AADL model described by such a class diagram of the AADL meta model maps into an XML document as follows:

- Concrete classes result in object instances in the AADL model.
- Labels of containment associations result in element tags.
- Class attributes result in XML attributes. Multi-valued attributes result in collections of XML elements.

- Reference associations within and across XML documents are represented by using the XML XPath construct as the reference notation [XPath 1999].

The XPath reference notation results in unnamed model objects in the reference path being recorded by their containment label, while named model objects are recorded by their containment label followed by the name as attribute, e.g., threadType[@name=GPS]. In case of a containment multiplicity of unnamed model objects the object is identified by the containment label and the index starting with zero as attribute, e.g., modeTransition[@index=0]. We always record these references as absolute references, i.e., references relative to the XML document root. For example, the following XPath references a thread type named *GPS* within the public section of a package specification named *edu::cmu::sei::XMIExample*.

```
compType="//aadlPackage[@name=edu::cmu::sei::XMIExample]/aadlPublic/thr
eadType[@name=GPS]".
```

References to model objects in the same XML document use the current AadlSpec object as root, as indicated by "//". References to model objects in other XML documents explicitly name the AadlSpec object. Its name is used to identify the name of the XML document that contains the referenced model. For example, the following XPath references the property definition *SafetyCriticality* that is declared in the property set *SEI* in the XML document *SEI*.

```
propertyDefinition="/aadlSpec[@name=SEI]/propertySet[@name=SEI]/propert
yDefinition[@name=SafetyCriticality]".
```

References encoded in the XPath reference notation are interpreted as URIs. The first element of the Xpath, if present, is interpreted like the URL portion of a URI, i.e., the AadlSpec name is mapped to a name in the file system to identify the file containing the XML document of the package. This allows the XML documents to be independent of the particular location of the file that contains the XML document.

Figure 5 shows an example AADL model that consists of a system type and a system implementation.

```
package edu::cmu::sei::XMIExample
public
    system GPS
    features
    init: in event port;
    signal: out data port GPS_Signal;
  end GPS;

  system implementation GPS.basic
  end GPS.Basic;

  data GPS_Signal
  end GPS_Signal;
end edu::cmu::sei::XMIExample;
```

Figure 5 An Example AADL Model

Figure 6 shows the XML document based for that AADL model. The highlighted attribute of a system implementation XML element shows the reference of the system implementation to its system type.

```
<?xml version="1.0" encoding="ASCII"?>
```



```
<core:AadlSpec xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:core="http://core" xmlns:property="http://property">
  <aadlPackage name="edu::cmu::sei::XMIExample">
    <aadlPublic>
      <systemType name="GPS">
        <features>
          <eventPort name="init"/>
          <dataPort name="signal" direction="out"
dataClassifier="//aadlPackage[@name=edu::cmu::sei::XMIExample]/aadlPubl
ic/dataType[@name=GPS_Signal]"/>
        </features>
      </systemType>
      <systemImpl name="GPS.basic"
compType="//aadlPackage[@name=edu::cmu::sei::XMIExample]/aadlPublic/sys
temType[@name=GPS]"/>
      <dataType name="GPS_Signal"/>
    </aadlPublic>
  </aadlPackage>
</core:AadlSpec>
```

Figure 6 XML Document of AADL Model

Figure 7 shows the class description for system types and system implementations in the AADL meta model. The association between the system type and system implementation is shown as bi-directional; the reference from the system type to the system implementation is defined as transient, i.e., it is not stored in the XML document. This reduces the size of XML documents, while at the same time provides for efficient access to objects in the AADL model when operated on by tools.

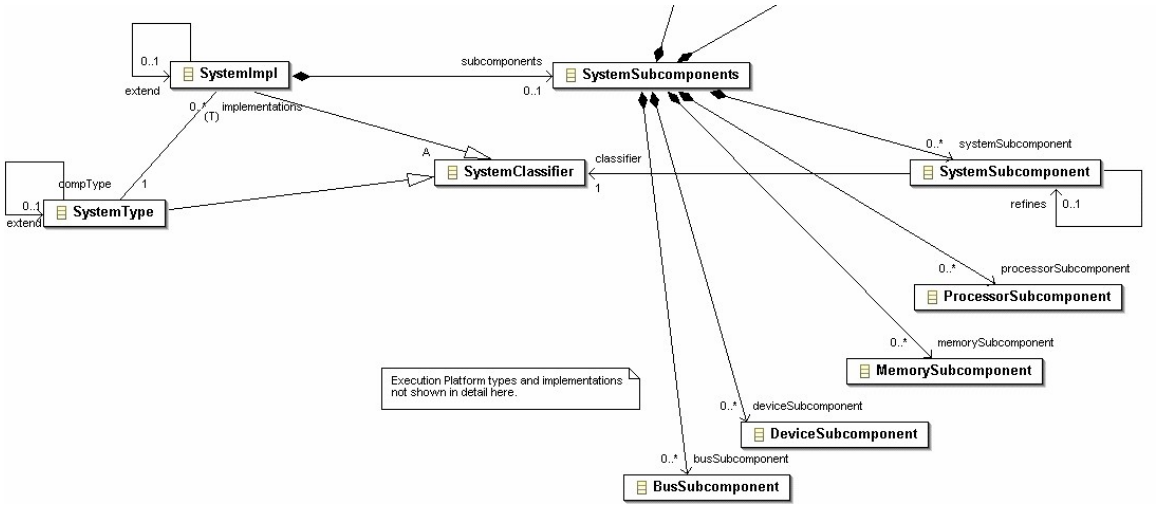

Figure 7 System Component Meta Model

Figure 8 shows the XMI specification of the system type and system implementation as it is generated by EMF from the Ecore meta model. Only the *compType* portion of the bi-directional reference association between the system type and the system implementation is specified for persistent storage. The *compType* reference association (shown as *xsd:element name="compType"*) is shown as being restricted to *SystemType* (shown as *type="component:SystemType"*). In other words, system implementations can only refer to

system types. The reference value is recorded as string valued attribute – in the example of Figure 6 shown in XPath format.

```
<xsd:complexType name="SystemType">
  <xsd:complexContent>
    <xsd:extension base="core:ComponentType">
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="extend" nillable="true"
type="component:SystemType"/>
        <xsd:element name="features" type="feature:SystemFeatures"/>
      </xsd:choice>
      <xsd:attribute name="extend" type="xsd:string"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="SystemType" type="component:SystemType"/>

<xsd:complexType name="SystemImpl">
  <xsd:complexContent>
    <xsd:extension base="core:ComponentImpl">
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="subcomponents"
type="component:SystemSubcomponents"/>
        <xsd:element name="compType" type="component:SystemType"/>
        <xsd:element name="extend" nillable="true"
type="component:SystemImpl"/>
      </xsd:choice>
      <xsd:attribute name="compType" type="xsd:string"/>
      <xsd:attribute name="extend" type="xsd:string"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="SystemImpl" type="component:SystemImpl"/>
```

Figure 8 XMI Meta Model of System Type & Implementation

Figure 9 shows the generated XML schema for the system implementation as it is generated by EMF from the Ecore meta model. The highlighted section shows that the reference association *compType* is recorded as a string. The restriction of the reference to system types is not explicitly recorded in the XML schema. This illustrates the richer expressive power of the XMI meta model specification over the XML schema specification.

```
<xsd:complexType name="SystemType">
  <xsd:complexContent>
    <xsd:extension base="core:ComponentType">
      <xsd:sequence>
        <xsd:element minOccurs="0" maxOccurs="1" name="features"
type="feature:SystemFeatures"/>
      </xsd:sequence>
      <xsd:attribute name="extend" type="xsd:string"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="SystemType" type="component:SystemType"/>

<xsd:complexType name="SystemImpl">
  <xsd:complexContent>
```



```
    <xsd:extension base="core:ComponentImpl">
      <xsd:sequence>
        <xsd:element minOccurs="0" maxOccurs="1" name="subcomponents"
type="component:SystemSubcomponents"/>
      </xsd:sequence>
      <xsd:attribute name="compType" type="xsd:string"/>
      <xsd:attribute name="extend" type="xsd:string"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="SystemImpl" type="component:SystemImpl"/>
```

Figure 9 XML Schema for System Type & Implementation

Annex C.2.3Validation of the AADL Meta Model

The AADL meta model has been validated through the construction of an Open Source AADL Tool Environment (OSATE). This environment has been built as a set of plug-ins to the Eclipse environment and is itself extensible. OSATE includes an AADL parser that translates textual AADL specifications into in-core declarative AADL models. Those declarative AADL models are persistently stored in XML according to the AADL meta model specification. OSATE also includes a semantic checker, various architecture analysis plug-ins, an XML to AADL text translator, an AADL object model editor, and an AADL graphical editing front-end.

Other specifications of the AADL meta model are possible. For example, the UML profile of the AADL includes a conceptual model of the AADL. In the case of the UML profile, these concept specifications are being kept consistent with the AADL meta model specification in this annex.

The conceptual model of AADL expressed by a collection of UML stereotypes and the Ecore meta model of the AADL differ primarily in how they represent the different forms of inheritance present in AADL. In AADL, component implementations inherit features from component types. Component types can inherit features from component types they extend. Similarly, component implementations that extend other component implementations inherit subcomponents, connections, and other elements. Finally, AADL has a rich set of inheritance rules for property values. The Ecore AADL meta model takes advantage of these different forms of inheritance by not replicating inherited objects of an AADL model. Instead, the AADL inheritance is part of the interpretation semantics of the AADL meta model. This has the benefit of compact models and XML documents. The interpretation of the different forms of AADL inheritance is realized through a library of AADL object model methods. In other words, tools that operate on AADL models can utilize these methods to access component type features from a component implementation or to retrieve a property for a component instance whose value has been declared for the component type of the instance.

The UML profile of the AADL offers the possibility of a UML-based XML interchange representation for AADL models. The XML schema and the XMI meta model specification for the AADL have been defined independent of such a UML-based XML interchange representation such that tool developers are not required to embrace UML and its interchange specification in order to support AADL-based architecture analysis and generation capabilities. The alignment of the conceptual model in the UML profile with the AADL meta model allows for a simple translation between the AADL XML documents and UML-based XML documents of AADL models.

Annex C.3 References

[EMF 2003] Eclipse Modeling Framework, F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, T. Grose. Addison-Wesley Eclipse Series, Aug. 2003. ISBN 0-13-142542-0.

[XMI 2003] XML Meta Interchange (XMI), Version 2.0, Object Management Group (OMG), May 2003. http://www.omg.org/technology/documents/formal/xmi.htm.

[XML 2001] XML Schema, Version 1.0, World Wide Web Consortium (W3C), May 2001. http://www.w3.org/XML/Schema.

[XPath 1999] XML Path Language (XPath) Version 1.0, World Wide Web Consortium (W3C), Nov 1999. http://www.w3.org/TR/xpath.

Annex C.4 Declarative AADL Model

Generation of textual AADL from the declarative AADL model has led to several requirements on the meta model.

- Preservation of AADL comments: A typical language processing front-end discards comments during parsing. The AADL parser preserves comments and associates them with the most appropriate model objects in the declarative AADL representation. During the regeneration of the textual representation these comments are included in the generated text.

- Preservation of declaration order: Although the AADL does not require a particular declaration order for component classifiers, it is necessary and desirable to preserve the declaration order. Preservation of the declaration order is necessary for features declared in port group types, because the declaration order of features is used to determine whether two port group types are inverses of each other. Preservation of declaration order is desirable when textual AADL that has been translated into the declarative AADL representation is regenerated from the object model.

Annex C.4.1Modular Meta Model

The AADL is a sizable language. Representation of the AADL meta model as a single unit would make it difficult to understand and maintain.

The AADL is extensible in that annex-specific sublanguages can be introduced. The modular meta model approach allows meta models for such sublanguages to be defined separately in Ecore and added to the core AADL meta model.

The meta model for the core AADL is divided into six meta model packages:

- *Core*: defines the concepts of component type, implementation, and subcomponent as abstractions, as well as packages and modes.

- *Component*: defines the concrete classes for the different categories of components, including the constraints on their subcomponents.

- *Feature*: defines the features of component types.

- *Connection*: defines the connections between component features.

- *Flow*: defines flow related elements of the AADL.

- *Property*: defines the elements for associating property values and for introducing new property types and properties via property sets.

Classes in one package of the meta model can be referenced by other meta model packages. This is done by preceding the Class name with the package name – separated by a double colon (“::”). Multiple graphical diagram views have been defined for some of the meta model packages in order to keep them readable.

The meta model for an initial AADL instance representation has been defined as a separate meta model package. Its details are discussed in section Annex C.5.

Annex C.4.2Hierarchy of Abstract and Concrete Classes

The AADL meta model is defined by a set of *abstract* and *concrete* classes. Abstract classes represent common characteristics for a set of subclasses. Concrete classes are instantiated as objects in the AADL model.

Figure 10 introduces the top-level set of abstract classes.

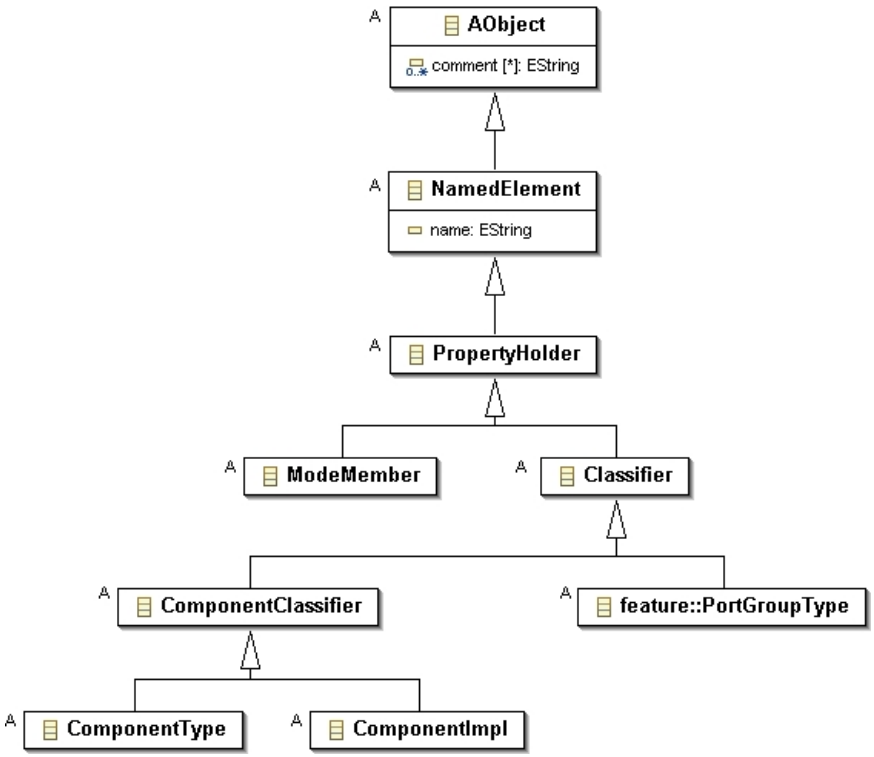

Figure 10 Abstract Class Hierarchy

The *AObject* class is the root class of all objects of an AADL model. This class introduces the *comment* as a multi-valued string attribute to hold AADL comments that are included in an AADL model.

The *NamedElement* class represents AADL model objects that are identified by name. It introduces a *name* attribute as a string value. For some AADL model objects the name is required, e.g., a component type, while for others the name is optional, e.g., a connection.

The *PropertyHolder* class represents those AADL model objects that can have AADL property association declarations. Note that for specific property definitions the legal owners of a property value for the defined property are specified by the property owner categories in the **applies to** clause of the property definition.

The *ModeMember* class represents those AADL model objects that are mode-specific. Mode-specific AADL model objects are those whose declarations are allowed to have **in mode** statements.

The *Classifier* class is a subclass of PropertyHolder and represents classifier declarations such as component classifiers and port group types. Classifiers are PropertyHolders, but not ModeMembers.

The *ComponentClassifier* class is a subclass of Classifier and represents component type, and component implementation declarations.

The *ComponentType* and *ComponentImpl* classes are subclasses of ComponentClassifier and represent component types and component implementations, respectively.

The *PortGroupType* class is a subclass of Classifier and represents port group types as definitions of the structure of port groups.

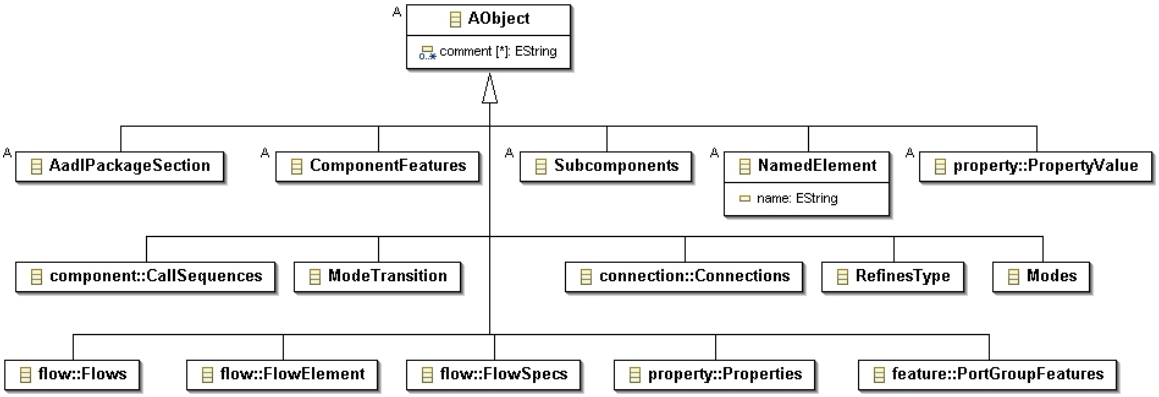

Figure 11 AObject Class and its Unnamed Subclasses

All AADL model objects have *AObject* as their abstract super class. This means that all AADL model objects can contain comments as attribute values. Figure 11 shows those AADL object model classes that are direct subclasses of *AObject*. These concrete subclasses are not subclasses of *NamedElement*, i.e., they instantiate to AADL model objects that are unnamed. The concrete subclasses of the shown abstract classes, with the exception of *NamedElement*, also instantiate to unnamed AADL model objects.

The classes ending named with a plural, such as FlowSpecs, Properties, or Connections, represent subclauses in a classifier. As explained previously, these are explicitly modeled to allow recording of optional subclauses and subclauses with no content, as specified by the reserved word **none**.

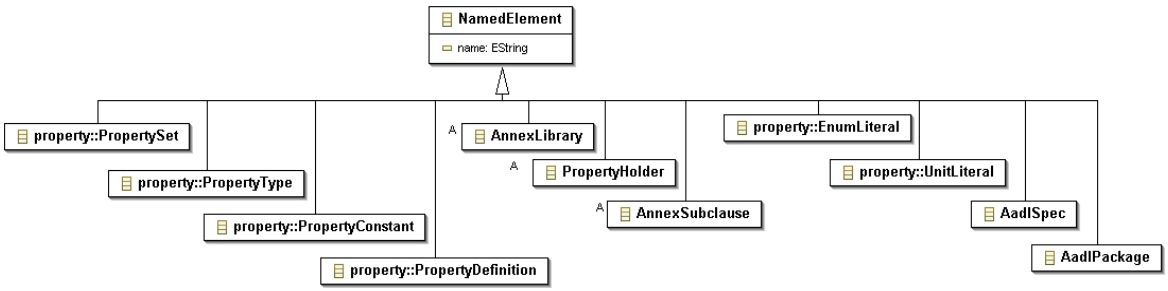

Figure 12 NamedElement Class and its Subclasses

Figure 12 shows the direct subclasses of the *NamedElement* class. They represent those AADL object model classes that have a name. Concrete subclasses of abstract subclasses of *NamedElement* are shown in subsequent figures.

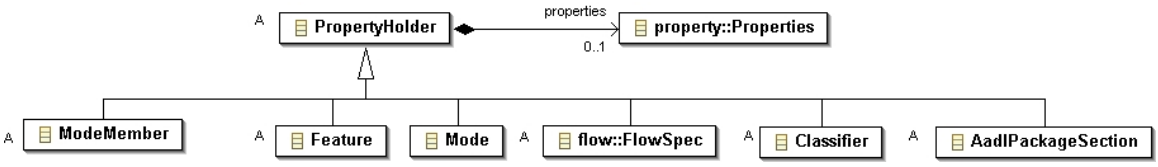

Figure 13 PropertyHolder Class and its Subclasses

Figure 13 shows those classes that can have properties associated with them, i.e., are subclasses of the *PropertyHolder* class. A property holder can contain zero or one Properties object that itself holds a collection of property associations.

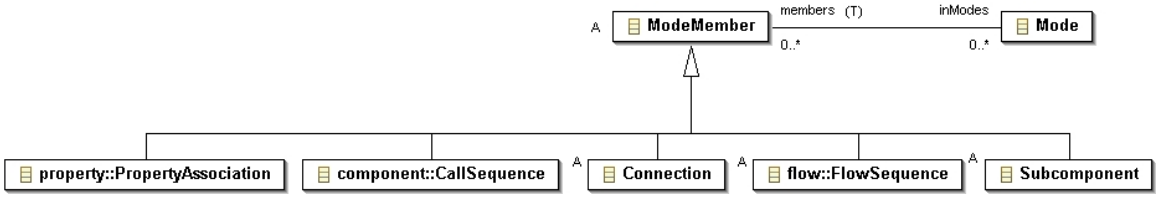

Figure 14 ModeMember Class and its Subclasses

Figure 14 shows the abstract class *ModeMember*. A ModeMember object can refer to multiple Mode objects via *inModes* and a Mode object refers to all objects that are *members* of it. The *members* reference is transient, i.e., not stored persistently in XML. Any object of concrete subclasses of this class can be declared to be a member of specific modes. The *ModeMember* class is a subclass of *PropertyHolder*. *PropertyAssociation* and *CallSequence* are concrete classes, while *Connection*, *FlowSequence*, and *Subcomponent* are abstract classes.

Annex C.4.3Component Types, Implementations, and Subcomponents

Component types can contain features, flow specifications, property associations, and annex subclauses. Figure 15 shows the content of a component type. The *ComponentType* class has containment associations to *Features*, *FlowSpecs*, and through inheritance to *AnnexSubclause* and *Properties*. A component type can contain zero or one object of each of these classes, and zero or more objects in the case of *AnnexSubclause*.

The *AnnexSubclause* abstract class represents annex subclauses that can be declared for any *Classifier*, i.e., for component types, component implementations, and port group types (see Annex C.4.8).

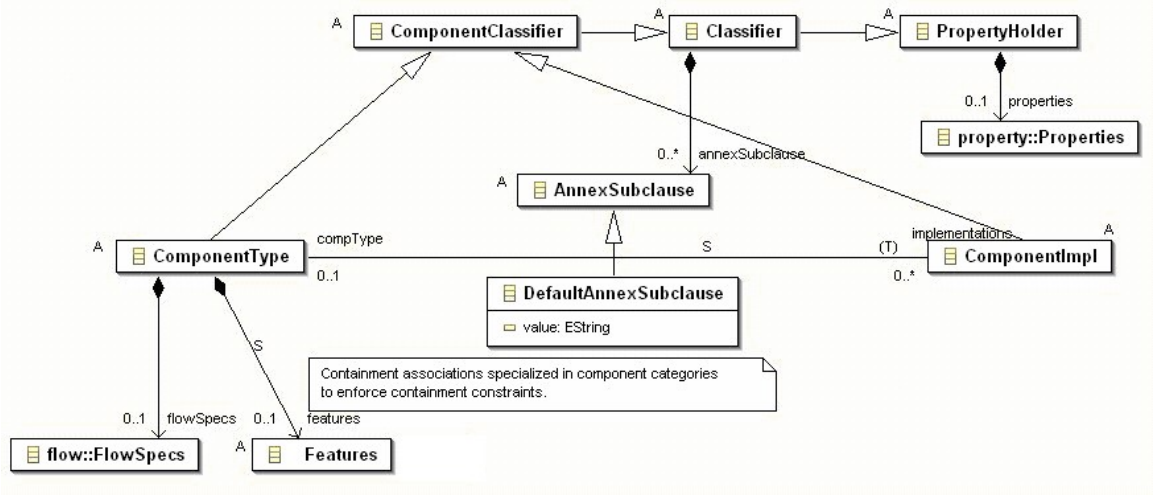

Figure 15 Contents of Component Types

Representing the subclauses as explicit objects has several advantages:

- The XML representation preserves the subclause structure - making XML documents quite readable.

- The existence of such a subclause object provides a distinction between the lack of such a subclause and a subclause with the **none** statement. This distinction is necessary to reproduce textual AADL.

The *Features* class represents the features subclause. It is an abstract class with concrete subclasses that record category-specific feature limitations. The association is shown as a containment association that is specialized for each of the component category-specific concrete subclasses of *Features* (see Annex C.4.7). Each specialization restricts the containment association for a specific component type to its respective *Features* subclass. The use of the same label for the containment association and its specializations permits the contained object to be retrieved by subclass-specific methods as well as a generic method of *ComponentType*.

The reference association from the *ComponentImpl* class to the *ComponentType* class is specialized for each of the category-specific concrete subclasses of *ComponentImpl* and *ComponentType*. Each specialization restricts the reference association to the respective category-specific subclass (see Annex C.4.4). The use of the same label for the two ends of the reference association and of its specializations permits the referenced objects to be retrieved by subclass-specific methods as well as a generic method of the abstract classes.

The *FlowSpecs* class represents the flow specifications subclause. It is a concrete class that contains individual flow specifications.

The *Properties* class represents the properties subclause. It is a concrete class that contains property associations. Its XML element tag is *properties*.

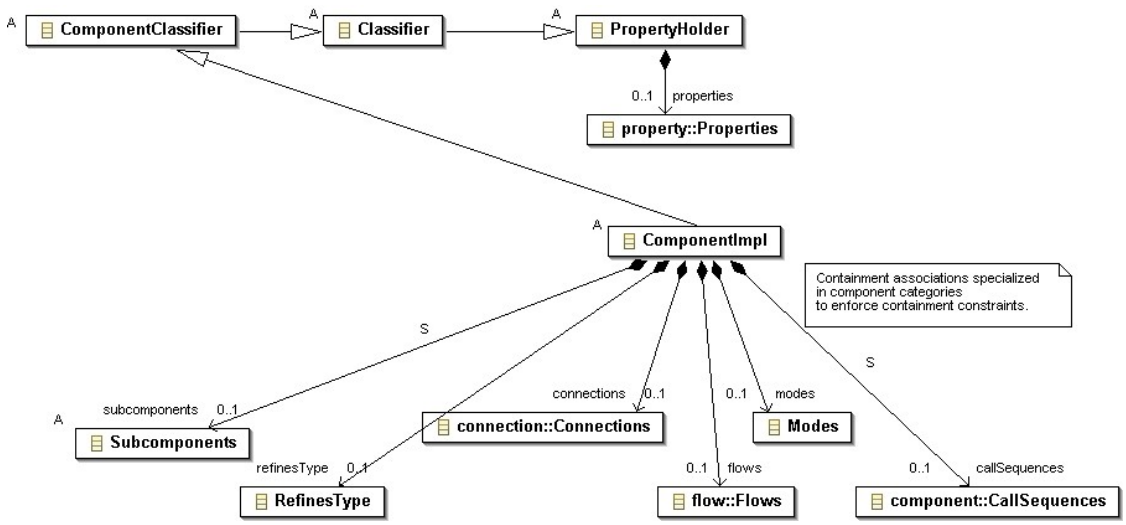

Figure 16 Contents of Component Implementations

Figure 16 shows the content of a component implementation. The *ComponentImpl* class has containment associations to the *Subcomponents*, *RefinesType*, *Connections*, *CallSequences*, *Flows*, *Modes* classes, and through inheritance to the *AnnexSubclause* (see Figure 15) and *Properties* classes. A component type can contain zero or one object of each of these classes, zero or more objects in the case of *AnnexSubclauses*.

The *Subcomponents* class represents the subcomponents subclause. It is an abstract class with concrete subclasses that record category-specific subcomponent limitations. The association is shown as a containment association that is specialized for each of the component category-specific concrete subclasses of *Subcomponents* (see Annex C.4.4). Each specialization restricts the containment association for a specific component implementation to its respective *Subcomponents* subclass. The use of the same label for the containment association and its specializations permits the contained object to be retrieved by subclass-specific methods as well as a generic method of *ComponentImpl*.

The *CallSequences* class represents the calls subclause. It is a concrete class which is contained in *ComponentImpl* by a containment association that is specialized for each of the component category-specific concrete subclasses of *ComponentImpl* (see Annex C.4.4). Each specialization determines whether the calls subclause is legal for a specific component category. The use of the same label for the containment association and its specializations permits the contained object to be retrieved by subclass-specific methods as well as a generic method of *ComponentImpl*.

The reference association between the component implementation and its component type represents a *component type inheritance* relationship. Similarly, the reference association between a subcomponent and its classifier represents a *component type inheritance* relationship. This means that the features defined in the component type are inherited by component implementations as well as by subcomponents. Component implementations cannot add new features. However, its connections can reference these inherited features as well as the features inherited by any contained subcomponent from their respective component classifier.

Such a component type inheritance concept is not a built-in concept of the Ecore meta modeling capability. Different from the UML profile for AADL, we have chosen not to replicate the features for the component implementation, which would require maintaining consistency of the replicated copies and would increase the size of the XML document considerably. Instead, AADL object model processing methods implement the inheritance by returning the appropriate set of features

when invoked on component implementations or subcomponents. Feature references, such as those of connections, identify the feature object in the component type and the context of the inherited feature being referenced (see Annex C.4.8).

Annex C.4.4Category-Specific Meta Model Classes

The AADL meta model has a separate class for each category-specific component type, component implementation, and subcomponent. These are concrete classes that get instantiated as objects of that class in the AADL object model.

These category-specific classes are used to model the fact that component implementations of a specific category must be implementations of a component type of the same category. They are used to model the fact that component types of a certain category can only contain certain features. They are also used to model the fact that component implementations of a certain category can only contain subcomponents of certain categories. The category-specific classes are used to model whether a component may or may not contain connections or call sequences. They may also be used to represent other restrictions expressed in the table of the Legality Rules section of each component category in the Standard.

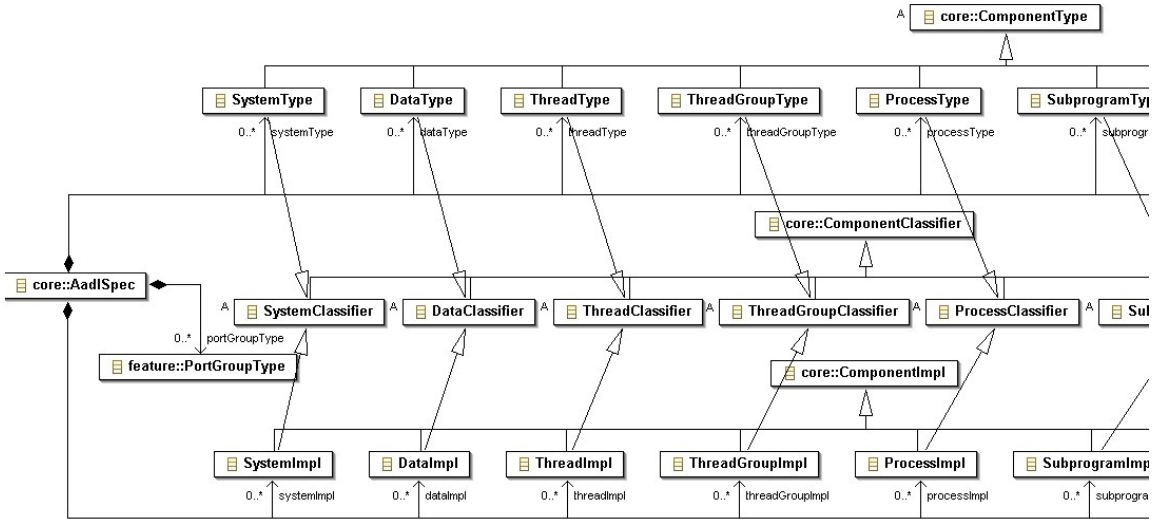

Figure 17 Category-Specific Classes

Figure 17 partially shows the category-specific classes and their subclass relationships in terms of component classifier, component type, and component implementation. In addition the figure shows that all category-specific classifiers, as well as port group types, can be declared in AADL specifications. A similar containment relationship is defined for AADL public and private package sections. The use of category-specific labels in the containment association results in XML element tags that reflect the category, e.g., the XML element tag <*systemType*> in Figure 6.

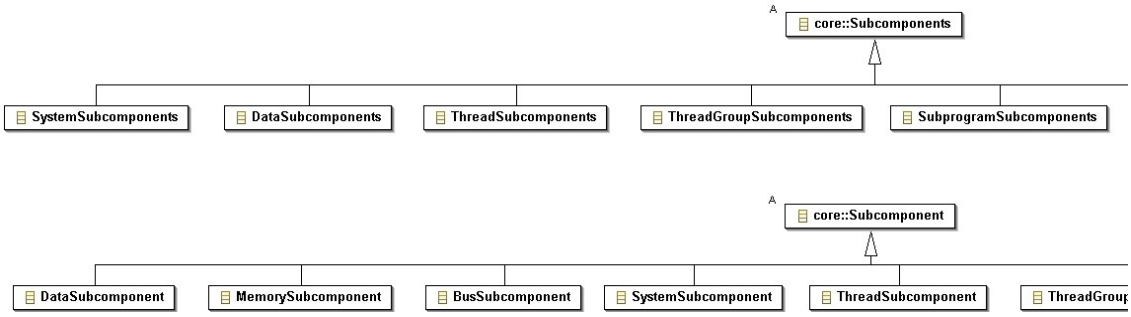

Figure 18 Category-specific Subcomponent Classes

Figure 18 partially shows the category-specific *Subcomponents* and *Subcomponent* classes as concrete subclasses. These classes are used to in Figure 19 to limit the categories of subcomponents that can be contained in each component category.

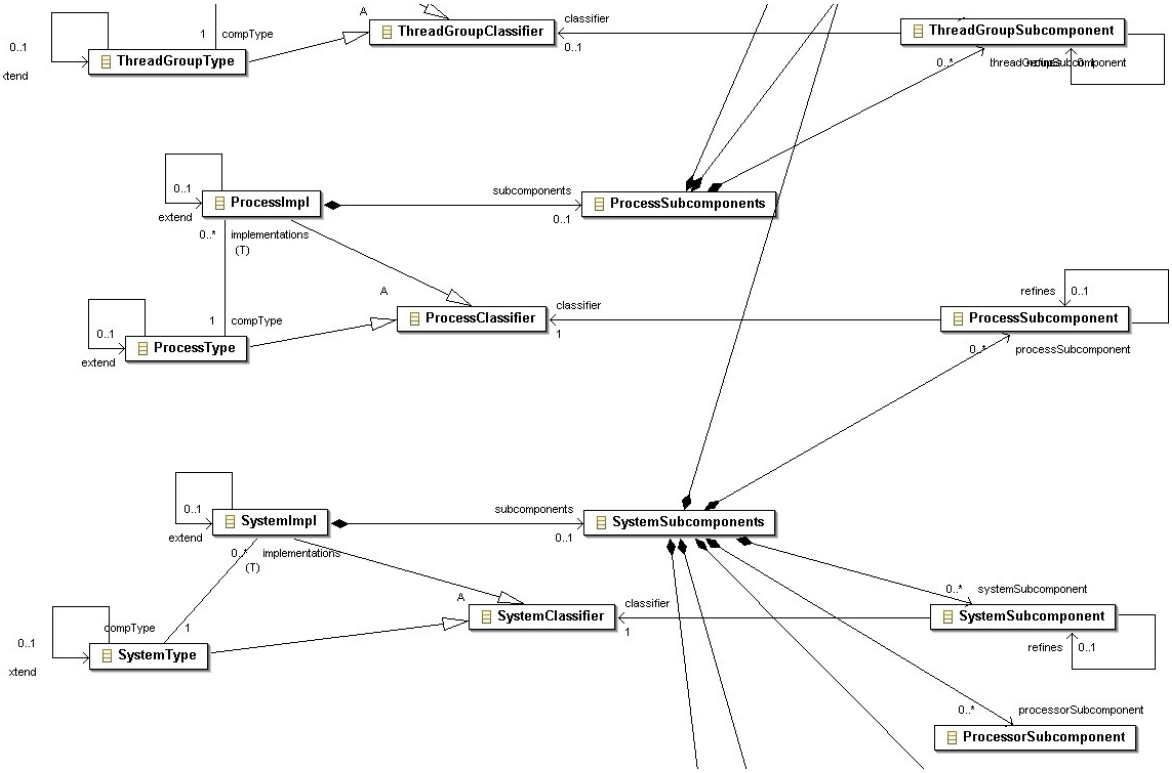

Figure 19 Component Category Specific Class Relationships

Figure 19 shows the containment and reference associations for a subset of category-specific classes. They are shown for the process and system categories. Both the *SystemImpl* class and *ProcessImpl* class has a *compType* reference association to its respective type class. This ensures that component implementations reference only component types of their own category. Each of these *compType* reference associations specializes the *compType* reference association of *ComponentImpl*. This means that the XML attribute name in both cases is *compType*, but the reference is restricted to the appropriate component type category.

Figure 19 reflects the constraints on the categories of subcomponents that are legal in a given component category implementation. For example, the system implementation contains a *SystemSubcomponents* class – the representative of the subcomponents subclause. Similarly,

the *SystemSubcomponents* class has containment associations to those category-specific Subcomponent classes that are legal for a system implementation.

Annex C.4.5Component Extension and Refinement

A component classifier or port group type declaration can extend another component classifier or port group type. This extension can refine previously declared model elements or add new model elements. This extends relationship is represented by an *extends* reference association of each category-specific component type or implementation or a port group type to itself. Figure 19 shows this reference relation for the process type, process implementation, system type, and system implementation.

The *extends* reference association represents an extension inheritance relationship. The model elements being inherited are not explicitly replicated in the persistent XML representation of the extension. Instead extension inheritance is used to keep the size of AADL object models and XML documents smaller.

Refinement of previously declared model elements in component types and implementations is represented by a *refines* reference association of each concrete subclass that is acceptable content to component types, component implementations, a port group types. Figure 19 shows the *refines* reference association for the *ProcessSubcomponent* and *SystemSubcomponent* class. The object representing the refined model element only contains the refinements in its persistent XML representation. Methods on the AADL object model provide access to the refined as well as the inherited attributes and associations of the refined object.

The *extends* reference association and the *refines* reference association are defined for each category to ensure that the references are to objects of the same class.

Annex C.4.6Category-Specific Containment and Ordering

AADL specifications, as well as public and private package sections, contain category-specific component classifiers (see Figure 17). The component category specific *Features* subclasses contain *Feature* subclasses representing data, event, and event data ports, port groups, parameters, data access, and bus access features (see Figure 21). The *Connections* class contains Connection subclasses that are specific to feature subclasses (see Figure 23). The *FlowSpecs* class contains *FlowSpec* subclasses that reflect flow sources, flow sinks, and flow paths (see Figure 26). Similarly, the *FlowSequence* class contains subclasses for flow source, sink, and path implementations, and for end-to-end flows.

These containments are represented by containment associations to the specific subclasses with the association name reflecting the subclass name. This results in XML element tags that correspond to the subclass name. These associations are specializations of a containment association defined between the container, e.g., the *AadlSpec* class, and the abstract super class of the contained classes, e.g., the *Classifier* class in Figure 32.

By default the contained objects are stored in an XML document grouped by their specialized containment associations. This does not maintain the declaration order found in a textual AADL model. An Ecore FeatureMap concept has been used to specify a desired containment ordering across all contained elements of specialized containment associations independent of their subclass. The result is an XML document with XML elements stored in the declaration order across the specializations and use of the XML element tags of the concrete subclasses.

Annex C.4.7Features

The abstract *Feature* class represents AADL features. This class is a subclass of *PropertyHolder* as well as *NamedElement*. An abstract subclass *AbstractPort* represents ports, port groups, and

parameters, i.e., all features that represent connection points for flow of data or events. *Port* itself is an abstract class representing data ports, event ports, and event data ports.

Concrete subclasses of the Feature class represent each of the different feature types in AADL. These classes are *DataPort, EventPort, EventDataPort, Parameter, Port Group, BusAccess*, and *DataAccess*. The class hierarchy of features is shown in Figure 20. The concept of an aggregate data port is not represented as a separate class, but through the `Aggregate_Data_Port` property on the port group. The figure also shows port group types, which will be elaborated in Annex C.4.8.

Port, parameter, and access features have a *direction* attribute. In the case of ports and parameters it is an enumeration value of type *PortDirection.* In the case of data or bus access it is an enumeration value of type *AccessDirection*.

All features except event ports have classifier references. These are represented by a single-valued reference association to the appropriate category-specific component classifier, or in the case of a port group to a port group type. This reference is optional and can refer to either a component type or component implementation to accommodate incomplete declarations in AADL.

Feature refinement is represented by an association of a feature class to itself to represent the fact that a feature refinement can only refine a feature of the same concrete class.

Component category specific feature classes have containment associations to each concrete feature class that is acceptable content for the component category according to the AADL legality rules. For application software components these are shown in Figure 21. For example, a data component type can only contain those features that are specified as by containment association for the *DataFeatures* class, i.e., *Subprogram* and *DataAccess*. *SoftwareFeatures* is an abstract class that defines the allowable features for its concrete subclasses for threads, thread groups, process, and system.

Figure 22 shows the containment constraints of features for the execution platform component categories.

The containment association name reflects the concrete feature class resulting in its use as an XML element tag for the feature. An ordering relationship is maintained across the concrete feature subclasses to record the declaration ordering (see Section Annex C.4.6).

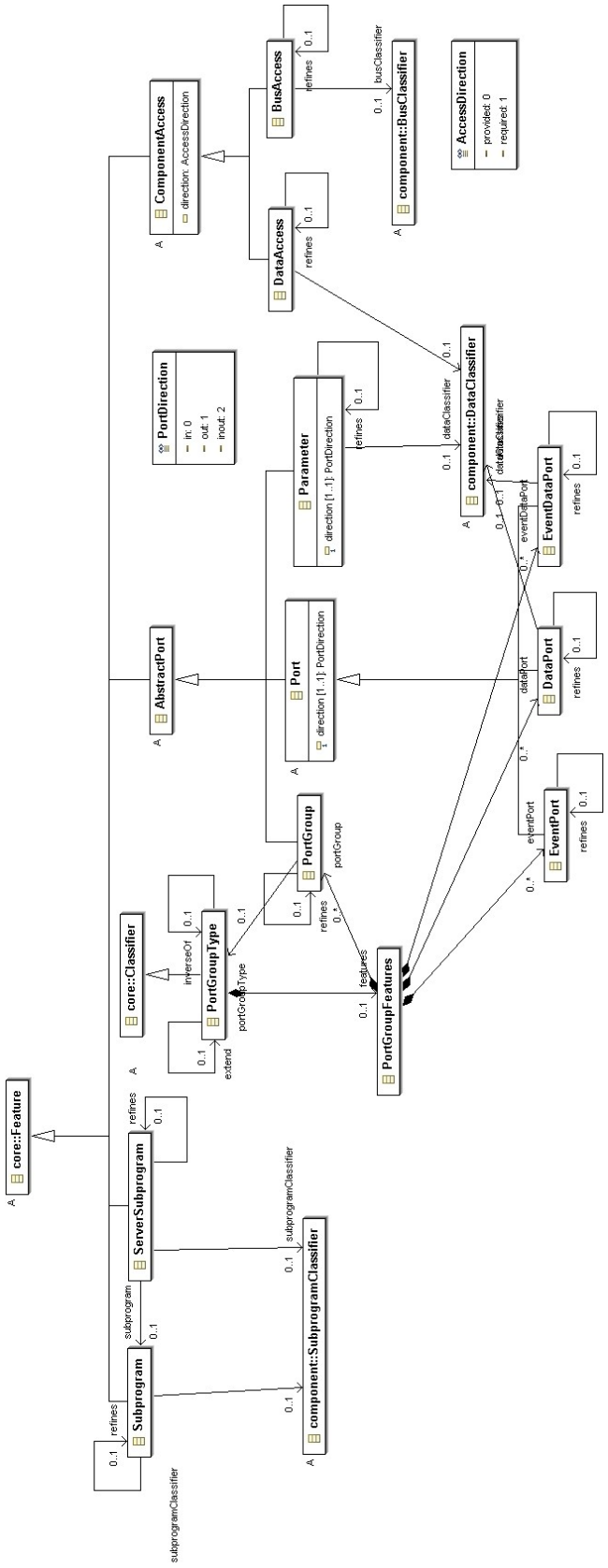

Figure 20 Features

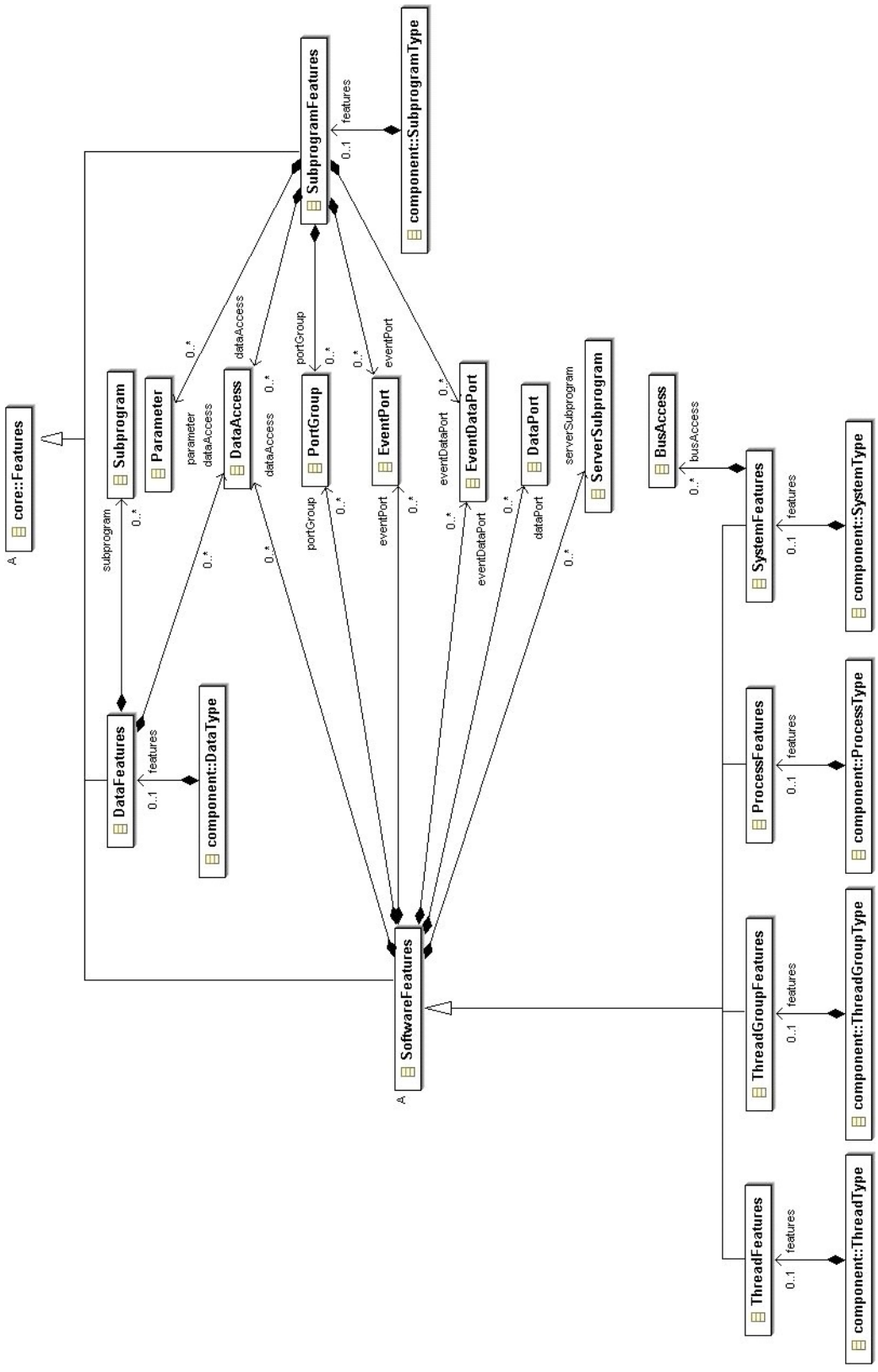

Figure 21 Software Category-Specific Feature Containment

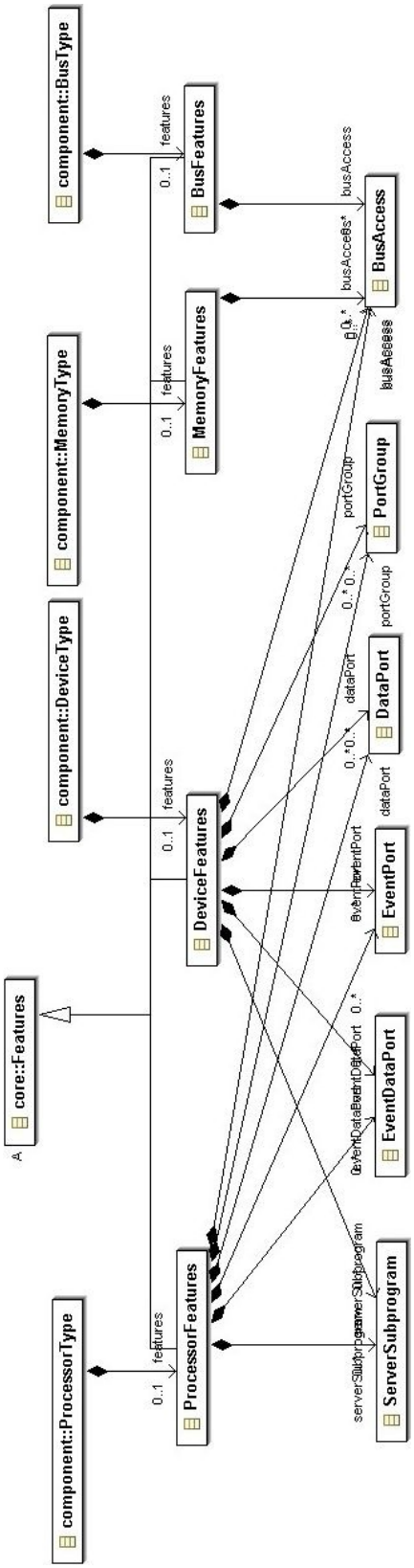

Figure 22 Execution Platform-Specific Feature Containment

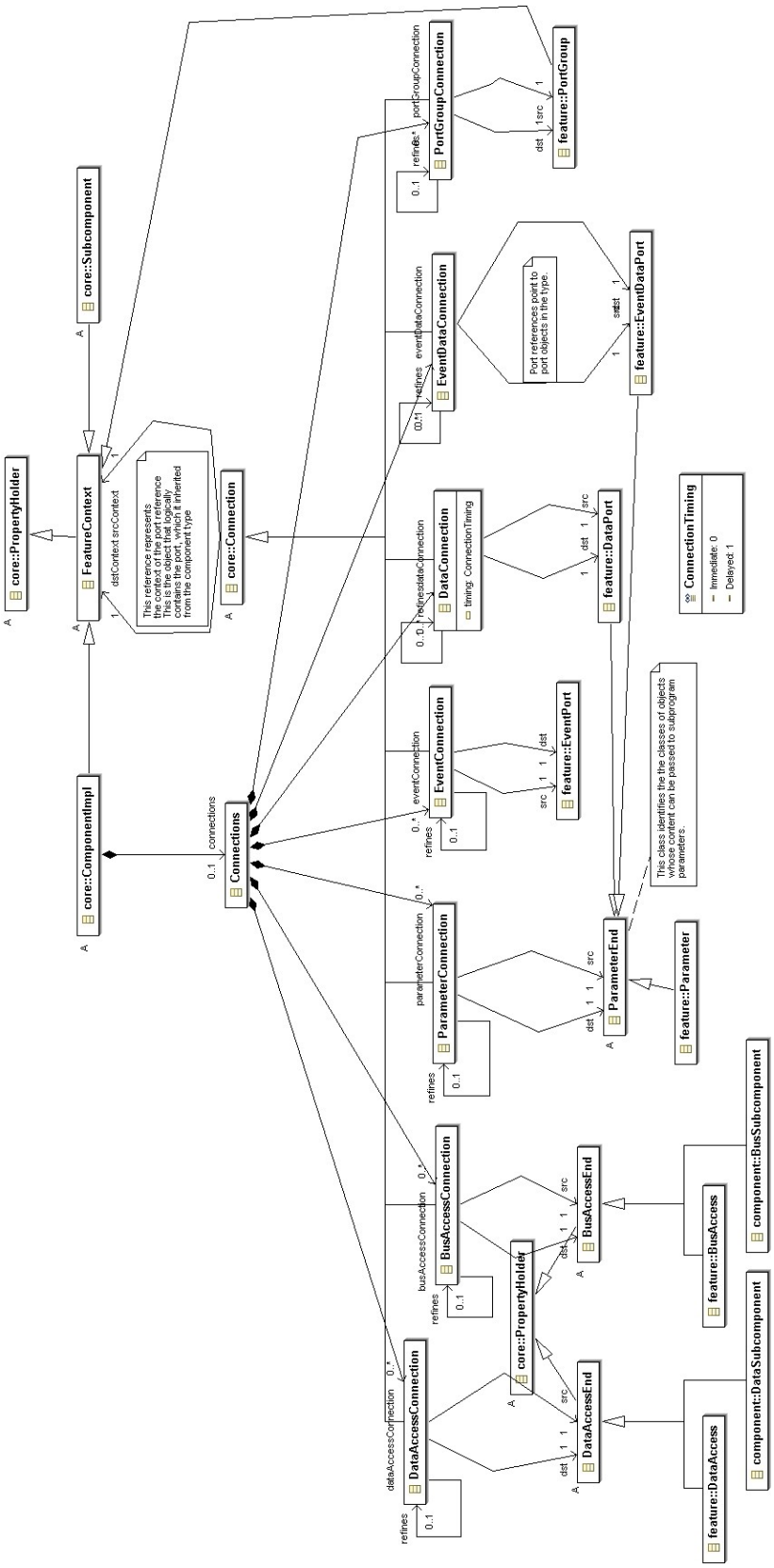

Figure 23 Port Connections

Annex C.4.8Port Group Types

The *PortGroupType* class is a concrete subclass of the *Classifier* class as illustrated in Figure 24. A *PortGroupType* class has an *extends* reference association and an *inverseOf* reference association to objects of the same class. Similar to a component type, it contains a features subclause that is represented by the *PortGroupFeatures* class. The *PortGroupFeatures* class has containment associations to the *DataPort*, *EventPort*, *EventDataPort*, and *PortGroup* classes. The specific class name is reflected in the association label resulting in its use as XML element tag.

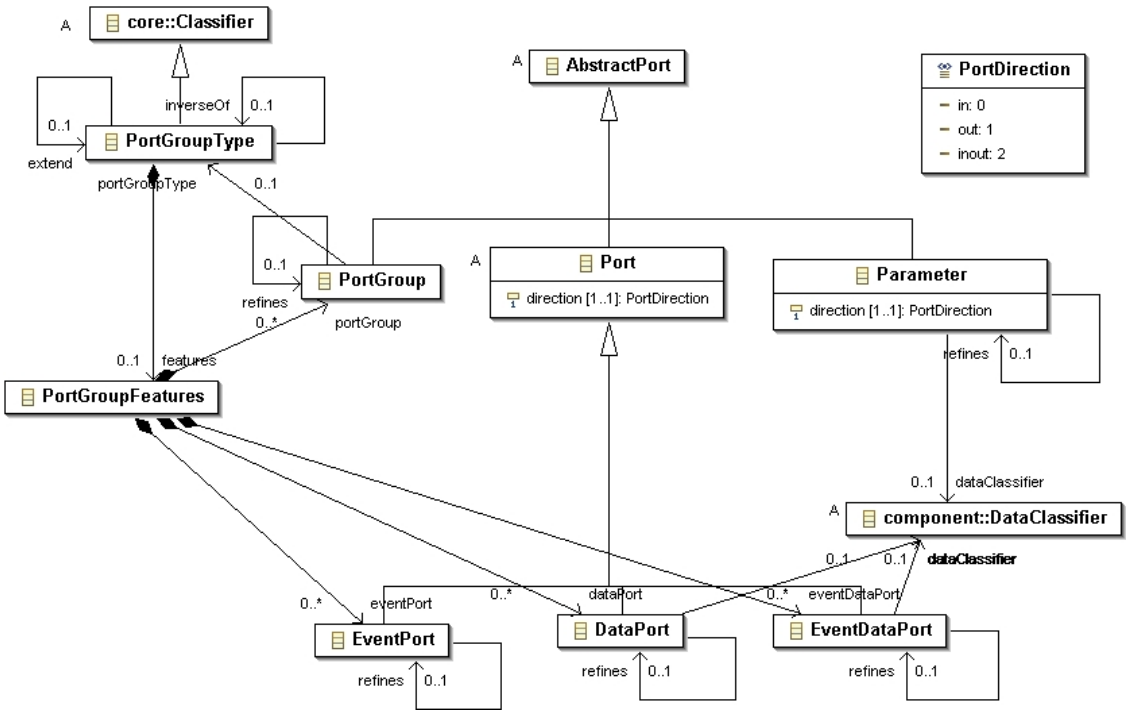

Figure 24 Port Group Type

Annex C.4.9Connections

The abstract *Connection* class represents a connection declaration of a connection between features of subcomponents, and between features of subcomponents and features of the enclosing component implementation. In the latter case, the connection may refer to a feature in the port group type of a port group in the enclosing component implementation. These classes are shown in Figure 23.

A connection has a source and a destination reference to features. These references must accommodate for the fact that the features of subcomponents, component implementations, and of port groups are inherited. This is achieved by a reference association to the feature as it is contained in the component type or port group type, and a reference association to the context in which the feature is referenced. This context is represented by the *FeatureContext* class - a super class that has *ComponentImpl*, *Subcomponent*, and *PortGroup* as subclasses.

The concrete classes that are subclasses of the *Connection* class represent the connection declaration for each of the different feature types. The source and destination reference associations are declared for each concrete class to reflect the fact that a connection of a certain type can only connect features of the same type. For example, a data port connection can only connect data ports. The concrete connection classes are *DataAccessConnection*,

BusAccessConnection, *ParameterConnection*, *DataConnection*, *EventConnection*, *EventDataConnection*, and *PortGroupConnection*.

All connection classes are subclasses of *ModeMember*, thus, have an *inModes* reference association to *Mode* objects. The *Connection* class also has an *inModeTransitions* reference association to *ModeTransition* objects (see Figure 28) to reflect the fact that connections can be specified to be active during mode transitions.

The source and destination reference associations are uni-directional, i.e., a connection refers to a port, but a port does not persistently maintain a reference to its connections. Methods simply determine the connections from and to subcomponent ports, i.e., connections to other subcomponents or the enclosing component implementation, from the set of connections of the enclosing component implementation. Similarly, the connections going from and to a port of a component implementation to one of its subcomponents, is the subset of connections of the component implementation that has the port as a source or destination reference.

A parameter connection cannot only refer to parameters, but also to data and event data ports. This is represented in the meta model by the *ParameterEnd* class with the *Parameter* class, *DataPort* and *EventDataPort* class as subclasses.

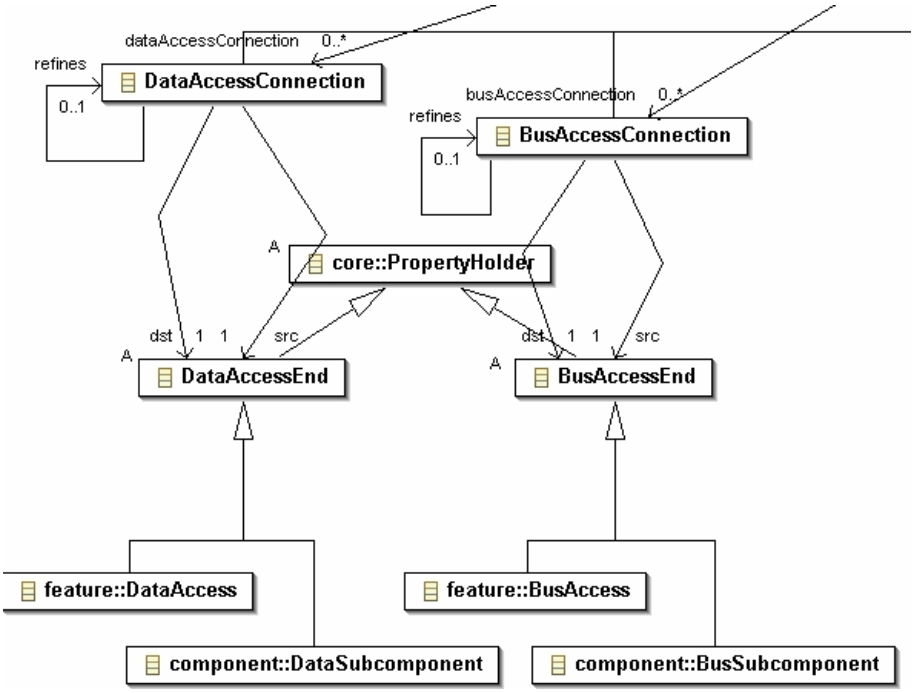

Figure 25 Access Connections

Figure 25 shows the classes and associations for access connections.

Bus and data access connections cannot only refer to access features as well as to data or bus subcomponent declarations. This is represented in the AADL meta model by the *DataAccessEnd* and *BusAccessEnd* classes with the appropriate access feature and category-specific subcomponent as subclasses.

The context of a bus or data access connection source and destination reference is filled in if the source or destination is a feature, i.e., the appropriate subcomponents and component implementations are referenced as context. If the reference is to a subcomponent the context is set to null.

Annex C.4.10 Flows

Flow specification declarations are represented by an abstract *FlowSpec* class, which itself is a subclass of the PropertyHolder class, and concrete classes for flow source (*FlowSourceSpec*), flow sink (*FlowSinkSpec)*, and flow path (*FlowPathSpec)*. These classes have references to members of the *AbstractPort* class in the same component type, represented by a source and destination reference association - as appropriate. A flow specification can also refer to an element of a port group. In that case, the port group is referenced as the context (*srcContext* or *dstContext*) of the source (*src*) or destination (*dst*) reference.

The abstract *FlowSequence* class represents both flow implementations and end-to-end flows. It is a subclass of the *ModeMember* class. A *FlowSequence* class contains a sequence of *FlowElement* objects that alternately represent a reference to a connection or a pair of references to a subcomponent and its flow specification.

Flow implementations are represented by the abstract *FlowImpl* subclass and the concrete subclasses *FlowSourceImpl*, *FlowSinkImpl*, and *FlowPathImpl*. Each of these concrete subclasses has a reference association to the respective flow specification class it implements. Flow implementations contain a sequence of *FlowElement* objects that alternately reference a *Connection* object or a *FlowSpec* object in a subcomponent. In other words, a *FlowElement* object either has a *connection* reference, or it has a *flowSpec* reference and a *flowContext* reference to a subcomponent. The *connection* reference must be to a port connection, i.e., to one of the concrete classes *DataConnection*, *EventConnection*, *EventDataConnection*, *PortGroupConnection*, or *ParameterConnection*. The originating source port and the final destination port of a flow implementation are identified by the flow specification being implemented, thus, are not explicitly recorded as a FlowElement object. Note that these ports are referenced by the first and last connection of the *FlowElement* sequence.

A *FlowSinkImpl* specifies a path from the flow sink spec port through zero or more connection and subcomponent flow specification pairs (*FlowElement*) with the last subcomponent flow specification referring to a flow sink. A *FlowSourceImpl* specifies a path from a subcomponent flow source specification represented by one *FlowElement* through zero or more connection and subcomponent/flow specification pairs (*FlowElement*) followed by a *FlowElement* containing the connection to the destination port of the flow source implementation. A *FlowPathImpl* specifies a path from the (incoming) source port to the (outgoing) destination port of the flow path spec through zero or more connection and subcomponent/flow specification pairs (*FlowElement*) ending with a *FlowElement* that contains the connection to the destination port of the flow path implementation.

End-to-end flows are represented by the concrete *EndToEndFlow* class through a sequence of *FlowElement* objects with the first referring to the originating subcomponent and flow specification pair and the remainder representing connection and subcomponent/flow specification pairs (*FlowElement*).

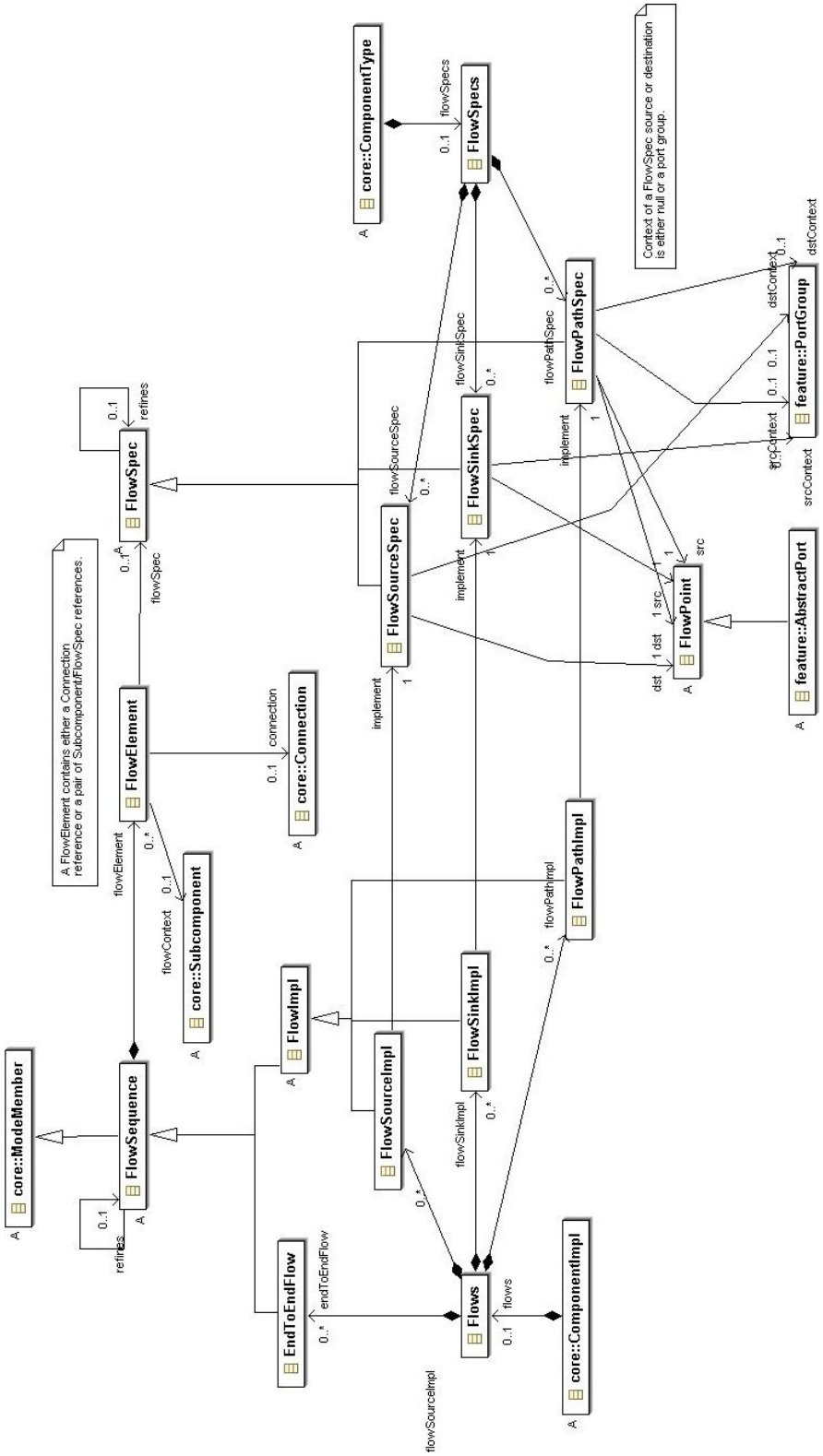

Figure 26 Flows

Annex C.4.11 Call Sequences

The concrete *CallSequence* class shown in Figure 27 represents a subprogram call sequence. It is a property holder and mode member. The *CallSequence* class has a multiplicity containment association to the *SubprogramSubcomponent* class to represent an individual subprogram call. Each such call has a name (the subcomponent name) and a subprogram classifier reference to identify the subprogram signature. Call sequences are permitted in instances of the *SubprogramImpl* class in the *ThreadImpl* class.

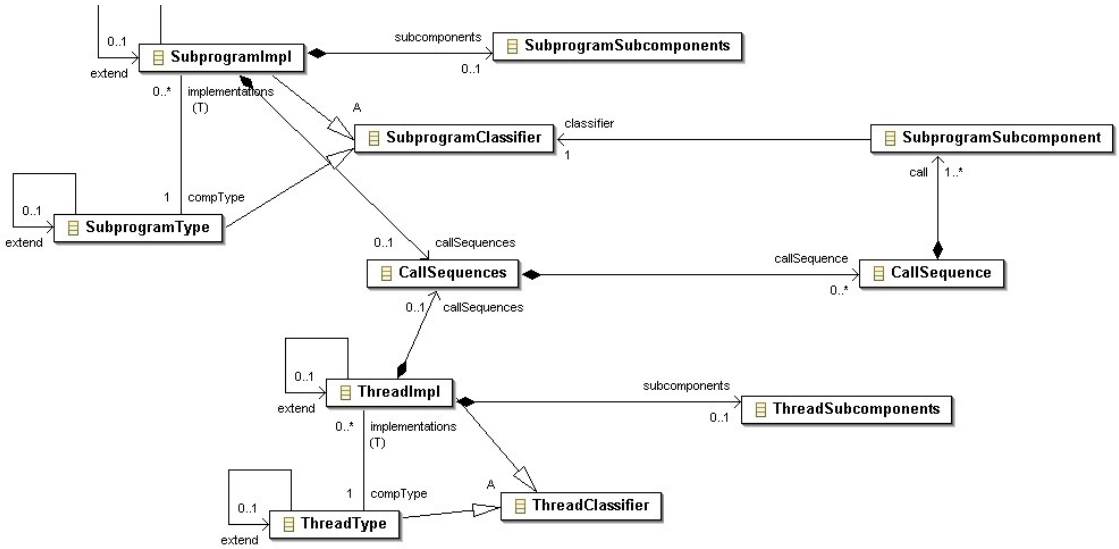

Figure 27 Subprogram Callsequences

Annex C.4.12 Modes and Mode Transitions

The *Modes* class shown in Figure 28 has a containment association with the *Mode* class representing an AADL mode and a containment association with the *ModeTransition* class. A *Mode* class can have a *refines* reference association to itself. The *ModeMember* abstract class has an *inModes* reference association to the *Mode* class; the *members* reference association in the opposite direction is a non-persistent association, i.e., not stored persistently in XML documents.

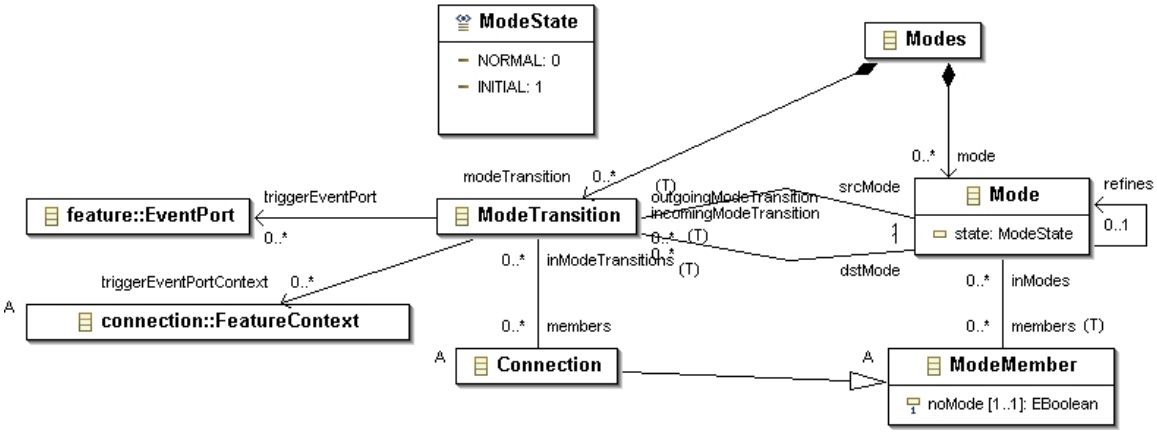

Figure 28 Modes and Mode Transitions

The *ModeTransition* class represents a mode transition. It has a *srcMode* and a *dstMode* reference association with the *Mode* class. The opposite reference associations *outgoingModeTransition* and *incomingModeTransition* are not persistent, i.e., not stored in XML documents. The abstract *Connection* class has an *inModeTransition* reference association to the *ModeTransition* class; the *members* reference association in the opposite direction is non-persistent, i.e., is not stored in XML documents. The Mode class has a state Attribute that records whether the mode is an *initial* mode, or normal mode.

Annex C.4.13 Property Associations

The *Properties* class can contain zero or more instances of the *PropertyAssociation* class (see Figure 29). The *PropertyAssociation* class is a concrete class. The *PropertyAssociation* class is not subclassed to reflect the type of the property value. Instead it contains zero or more instances of concrete subclasses of the *PropertyValue* class. The multiplicy allows lists of property values to be represented. A *PropertyAssociation* class also has a reference association to the concrete *PropertyDefinition* class. In other words, multiple property associations of the same property point to the same property name declaration (property definition). A *PropertyDefinition* class has a *propertyTypeReference* reference association to a separately declared named property type or a *propertyType* containment association to concrete subclasses of the abstract *PropertyType* class representing an unnamed property type declared as part of the property definition.

The PropertyAssociation class is not subclassed to reflect different property types, the reason being that user-defined property types can be introduced through property sets (see Annex C.4.16). This means that matching of the *PropertyValue* type with the type specified for the property definition is not enforced by the XML schema or the XMI meta model; it needs to be checked by the semantic checker instead.

The *PropertyAssociation* class has an *appliesTo* reference association of zero or more to the *PropertyHolder* class to reflect the sequence of concrete *PropertyHolder* subclasses. This sequence represents the path to a contained model element to which the property association applies. The *PropertyAssociation* class has an *inBinding* reference association of zero or more to the *ComponentClassifier* class to reflect the set of concrete execution platform *ComponentClassifier* subclasses for the inbinding statement of the property association.

The *PropertyAssociation* class is a subclass of *ModeMember*. This means that it also has a multiplicity reference association named *inModes* to the *Mode* class. This means that a separate *PropertyAssociation* object is created for different property association declarations of the same property with values for different modes. For example,

```
Latency => 10 ms in modes (normal);
Latency => 8 ms in modes (highspeed);
```

translates into two *PropertyAssociation* objects, each with a property value object.

PropertyAssociation has the constraint that although it is a *ModeMember* and as such a subclass of *PropertyHolder*, it cannot contain property association declarations. This reflects the fact that properties cannot have properties themselves.

The *access*, *append*, and *constant* attributes of the *PropertyAssociation* class reflect whether the property association is declared as an access property association, whether to append the value(s), and whether the property association should not be further overwritten. The *derived* attribute is used to indicate whether a property association attached to an instance object has been derived (cached) from a property association in the declarative model (for more on instance object see Annex C.5). A property association that is marked as not *derived* is only associated with the instance model. Such an instance model specific property association is used if the result of an analysis applies only to the instance model.

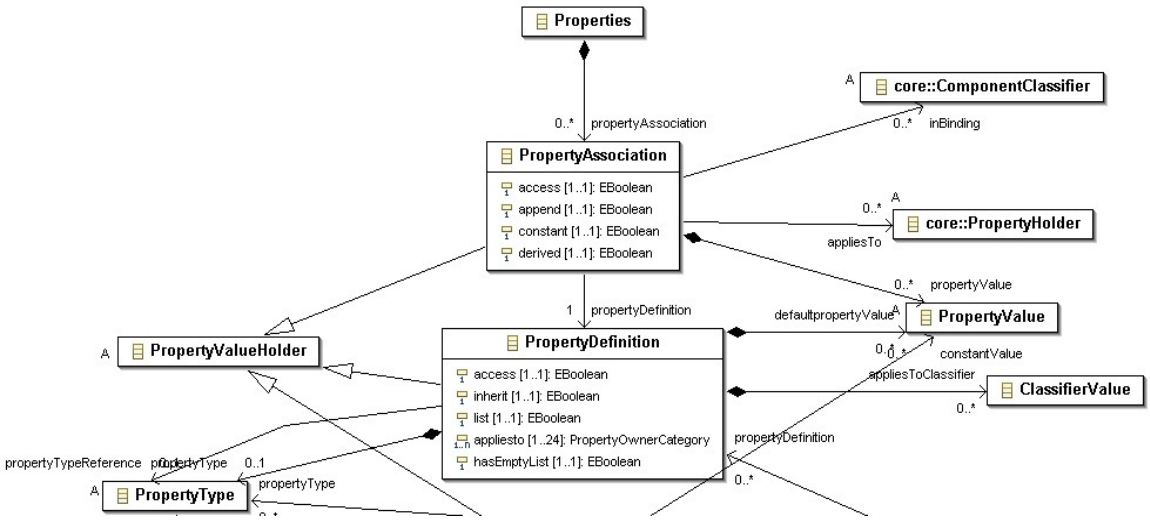

Figure 29 Property Associations

Annex C.4.14 Property Values

The AADL supports a number of different property values that correspond to the property type of the property definition referenced by a property association. The property value is represented by the abstract *PropertyValue* class. Concrete subclasses represent values of specific types. The value types *StringValue*, *IntegerValue*, *RealValue*, *IntegerRangeValue*, *RealRangeValue*, *EnumValue*, *ClassifierValue*, and *ReferenceValue* are shown in Figure 30. *BooleanValue* and *PropertyReference* are shown in Figure 31.

The common super class to *IntegerValue* and *RealValue* called *NumberValue* has a reference association called *unitLiteral* to a unit literal defined as part of an enumeration type. An *IntegerRangeValue* and *RealRangeValue* object contains a minimum value, a maximum value, and an optional delta value, represented by appropriate containment associations defined for a common super class called *RangeValue*. These containment associations refer to a class called *NumberOrPropertyReference*. This class is a super class of *NumberValue* and of *PropertyReference* and represents the fact that the value can be expressed as an actual value, or as a reference to a property constant.

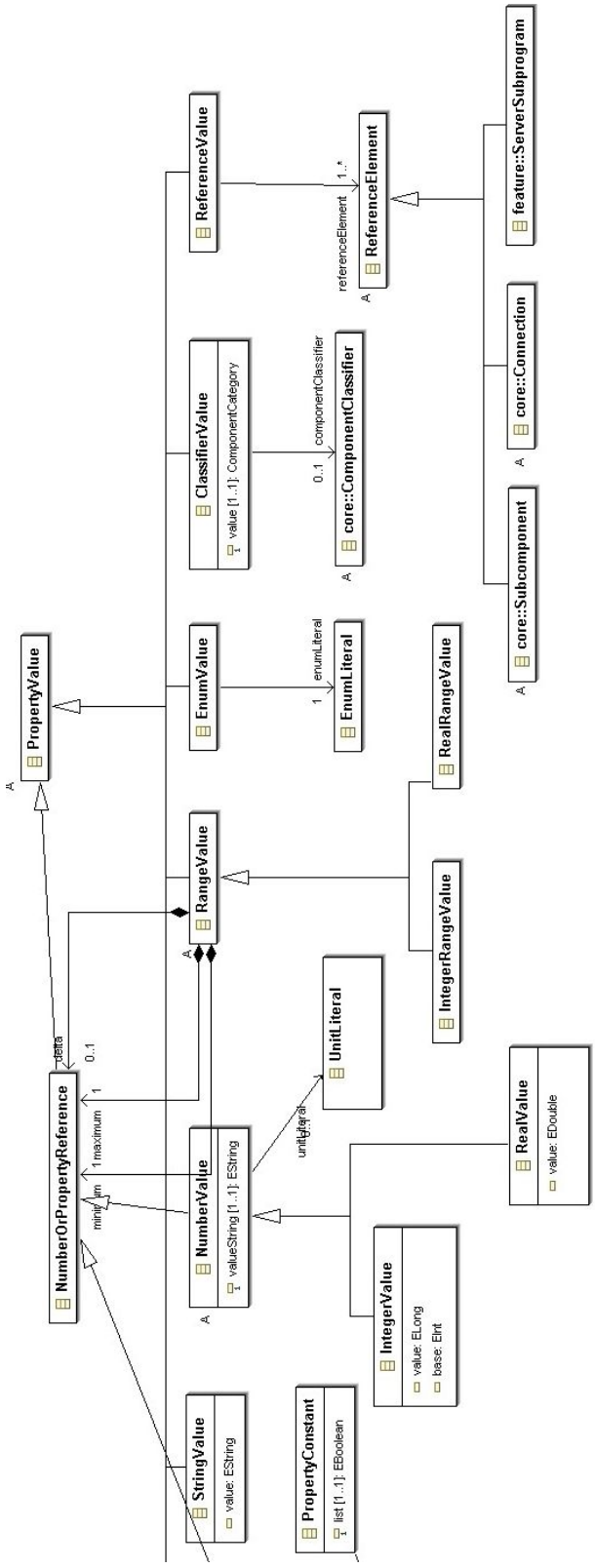

Figure 30 Property Values

NumberValue objects have a *valueString* attribute to represent the value as a string. This is done to preserve the particular representation chosen in the textual AADL representation. For example, the integer value 10,000 may also be represented as 10e5. In addition, the actual value, and in the case of integers the base, are also stored as *value* and *base* attributes.

Enumeration values are represented by objects that have a reference association to an enumeration literal. Classifier values are represented by the concrete *ClassifierValue* class, which records the classifier's component category in an attribute and an optional classifier reference with a reference association. Acceptable enumeration values for the *componentCategory* attribute are defined in Figure 33. The *ReferenceValue* class represents reference values to elements in the AADL model. Those references are recorded as a sequence of references to subcomponents terminating in a subcomponent, connection, or server subprogram. The sequence is realized by a multiplicity reference association and represents a path to the referenced element. The elements that can be referenced, are represented by the abstract class *ReferenceElement*.

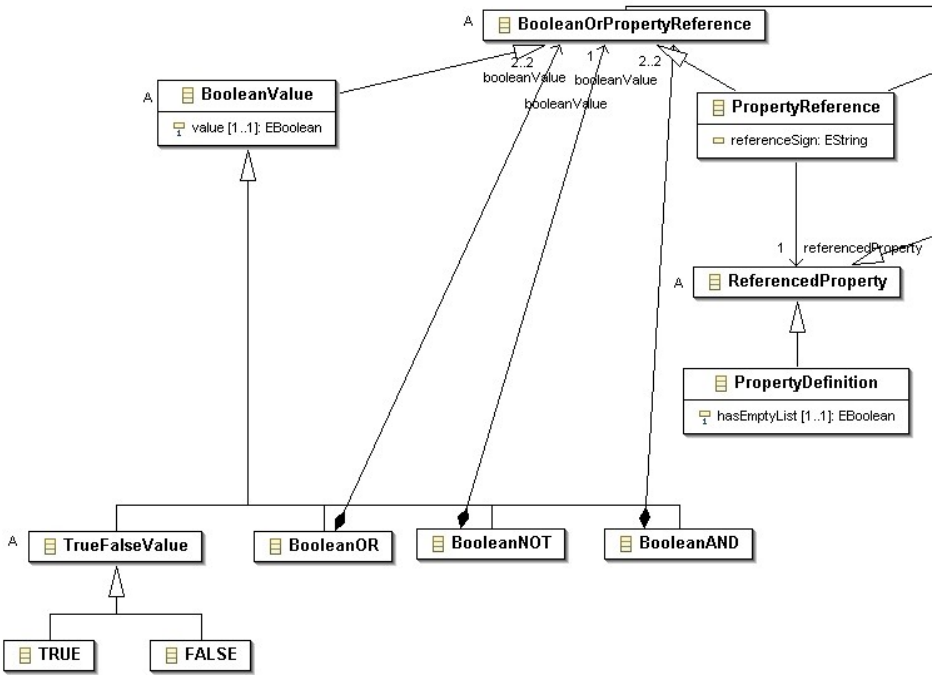

Figure 31 Boolean Property Expression

The AADL supports Boolean expressions as Boolean values. This is represented by the concrete subclasses *TRUE*, *FALSE*, *BooleanOR*, *BooleanAND*, and *BooleanNOT*. The three classes representing the Boolean operators have containment associations pointing to the *BooleanOrPropertyReference* class to reflect the fact that either Boolean values or references to other properties or property constants are permitted. The precedence order of the Boolean operators, as defined in Section 10.4 (p.164 of AS5506), is reflected in the containment hierarchy of the Boolean operator objects.

Property values can be expressed as references to a property constant or another property using the **value** construct. Such references are represented by the *PropertyReference* class. In the case of references to number property constants, the reference can include a sign, which is recorded as a *sign* attribute. A null value indicates no sign, the strings "+" and "-" represent the plus or minus sign. A *PropertyReference* can refer to a *ReferencedProperty* class, i.e., a *PropertyDefinition* or a *PropertyConstant* class. Property constants and property definitions, also known as Property name declarations, are described in Annex C.4.16.

Annex C.4.15 AADL Packages

AADL packages are represented by the concrete *AadlPackage* class, which is a NamedElement subclass. This class is the root class for AADL packages stored as separate XML documents. Figure 32 shows the containment associations for AADL specifications and packages.

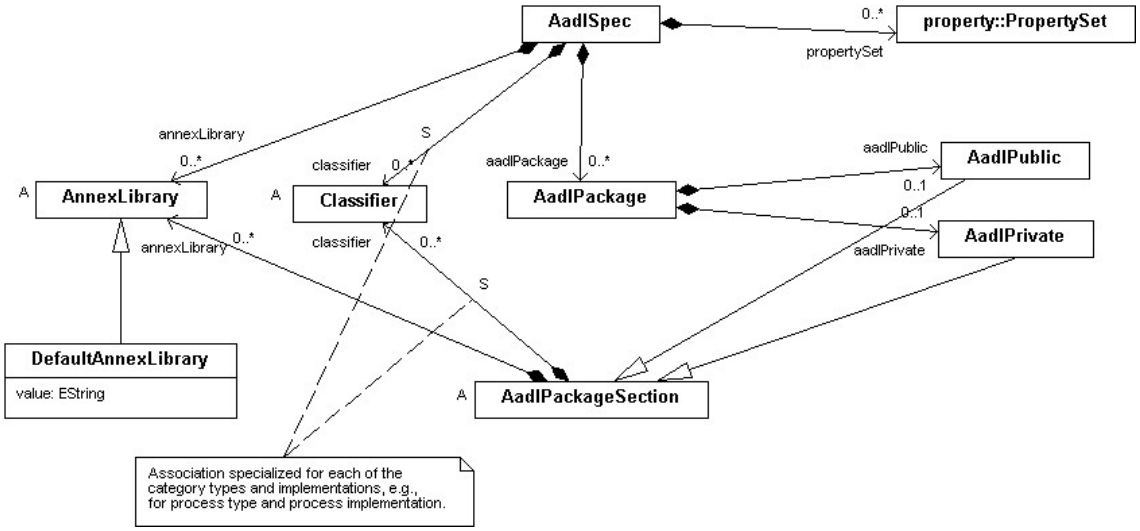

Figure 32 AADL Specifications and Packages

AADL packages contain at most one public and one private package section. These sections are represented by an *AadlPublic* and an *AadlPrivate* class, which are concrete subclasses of the *AadlPackageSection* class.

The *AadlPackageSection* class is defined as a subclass of the *PropertyHolder* class to represent the fact that public and private package sections can contain a Properties subclause.

The *AadlPackageSection* class is shown to contain zero or more of *Classifier* and *AnnexLibrary*. The *Classifier* containment association is specialized to the concrete classifier class using its name as the association label. This results in XML element tags for component classifiers that identify the category component type or implementation. An ordering relationship is maintained across these subclasses to represent a user-specified declaration order (see also Annex C.4.6). The ordered set of classifiers is returned by the method implementing the classifier containment association.

Annex C.4.16 Property Sets, Types, Constants, and Definitions

The AADL supports introduction of user-defined properties, property constants, and property types through property sets.

The *PropertySet* class has multiplicity containment associations to the *PropertyType*, *PropertyConstant*, and *PropertyDefinition* classes. The abstract *PropertyType* class represents property type declarations. Instances of the concrete *PropertyType* subclasses, i.e., *UnitsType*, *Aadlreal* and *Aadlinteger* and their common super class *NumberType*, *RangeType*, *Aadlstring*, *Aadlboolean*, *EnumType*, *ReferenceType*, and *ClassifierType,* represent property type declarations. Their name attribute records the name of the newly declared property type. The abstract *PropertyValueHolder* class represents all classes that can contain property values, i.e., property definitions, property constants, and property associations.

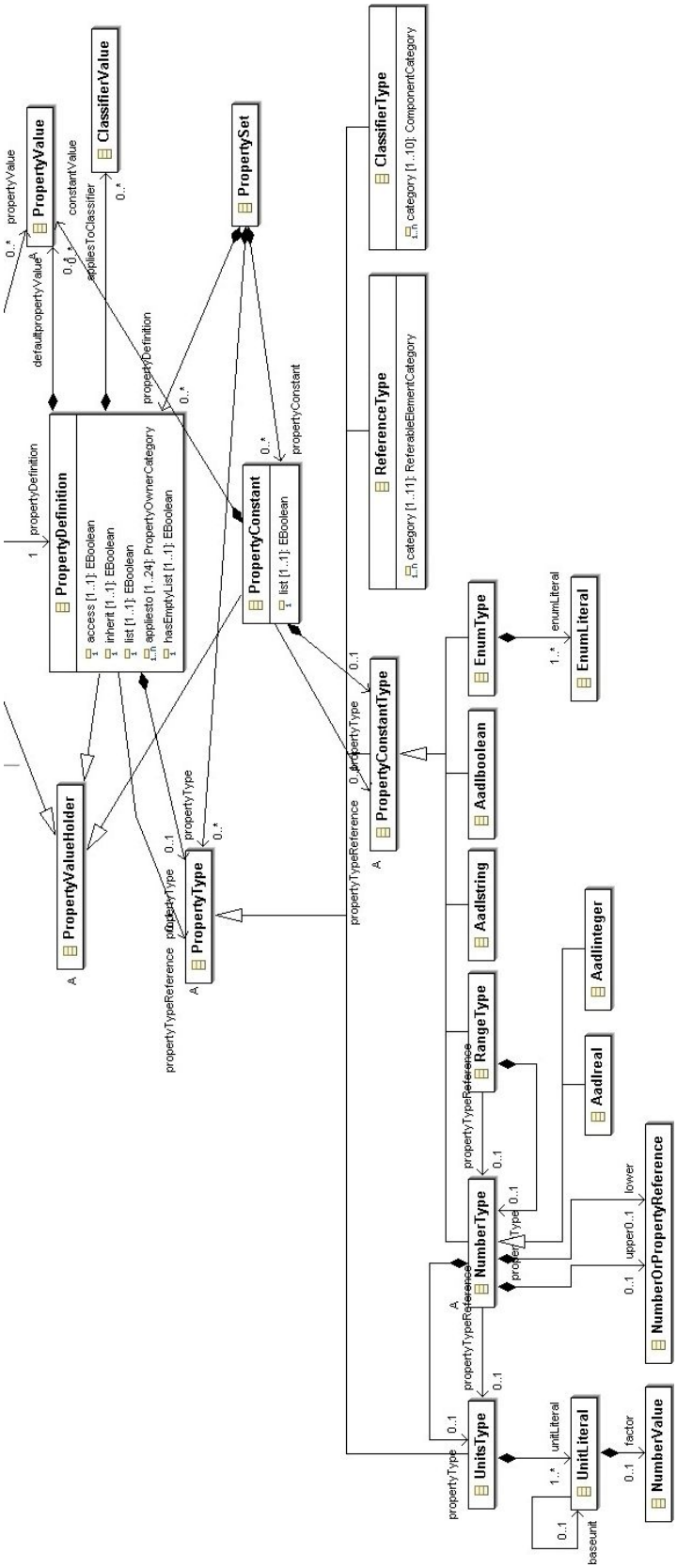

Figure 33 Property Sets, Types, Constants, and Definitions

The *UnitsType* class contains instances of the *UnitLiteral* class. Unit literals can be defined with conversion factors relative to other unit literals. This is reflected in the *baseunit* reference association and the containment association *factor* to a *NumberValue*.

The *NumberType* class has containment associations called *lower* and *upper* to the *NumberOrPropertyReference* class to represent the lower and upper bound on values of this property type. The NumberType class has a *propertyTypeReference* reference association and and a *propertyType* containment association to *UnitsType* to represent a reference to a declared units type or to represent an unnamed units type declared as part of the *Aadlinteger* or *Aadlreal* declaration. Either the *propertyTypeReference* association or the *propertyType* association is set.

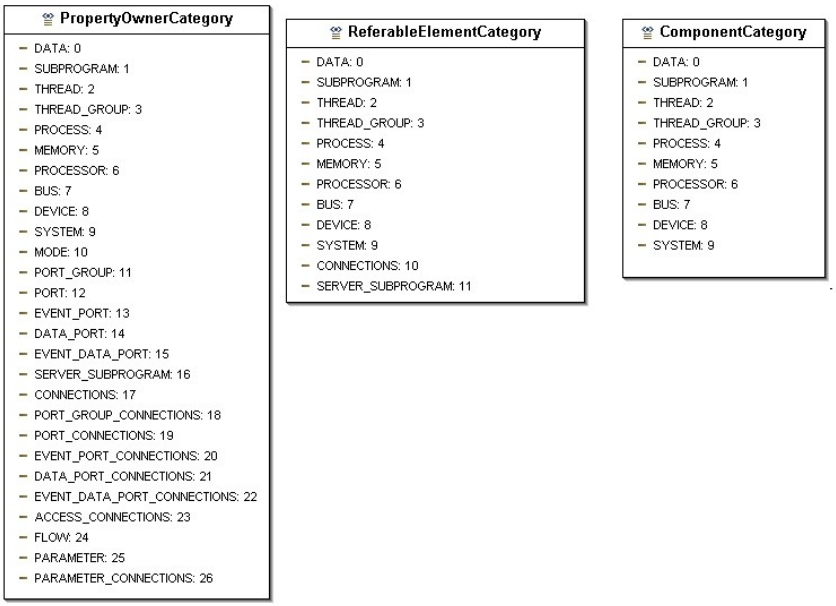

Figure 34 Enumerations in the Property Meta Model

Property constants can be declared for a subset of the property types. This subset is defined by the abstract class *PropertyConstantType*. The *PropertyConstant* class refers to a previously declared property type or contains an unnamed property type declaration, using the *propertyTypeReference* or *propertyType* association. A property constant also contains zero or more *PropertyValue* objects to represent a single property value or a list of values including an empty list.

Property name declarations are represented by the *PropertyDefinition* class. As mentioned in Annex C.4.13, this class has a reference association to a named property type, whose PropertyType object is contained in a PropertySet object, or it has a containment association to a PropertyType object that represents an unnamed property type declared as part of the property definition. An example of an unnamed property type in a property name declaration is `Acceleration: aadlreal units ( mps2 );` A PropertyDefinition class also has a multiplicity containment association to the *PropertyValue* class to represent the default value, and a containment association to the *ClassifierValue* class to represent the optional set of classifiers within a specified component category that the property can use – as specified in the **applies to** portion of the property name declaration.

The PropertyDefinition class has several attributes to record whether certain reserved words are part of the property name declaration. They are *access*, *inherit*, and *list*. In addition there is the attribute called *appliesto* that records the categories of AADL model objects for which the

property is applicable. This attribute has a multiplicity of up to the size of the *PropertyOwnerCategory* enumeration. This enumeration and others used in the property meta model package are shown in Figure 34. The *hasEmptyList* attribute is used to record the difference between no default value and a default value of an empty list for property name declarations that can hold lists of values.

Annex C.4.17 Annex Libraries and Annex Subclauses

The SAE AADL standard supports extension of the core language through annex library declarations and annex subclauses for different annex sublanguages. These are represented by the abstract classes *AnnexLibrary* (shown in Figure 32) and *AnnexSubclause* (shown in Figure 15). These classes are subclasses of NamedElement with the annex name recorded in the *name* attribute.

These classes have concrete subclasses *DefaultAnnexLibrary* and *DefaultAnnexSubclause*. Those default classes record the content of an annex library or annex subclause declaration as a string. This allows annex sublanguage expression to be preserved without parsing them.

If an annex sublanguage is introduced as a meta model extension in the form of an Ecore package, then the root expression of the sublanguage is introduced as a new subclass of *AnnexLibrary* and *AnnexSubclause*. In this case, a sublanguage parser can convert the sublanguage text into an object representation that is a natural extension of the core AADL object model.

Annex C.5 AADL Instance Model

The *Instance* meta model package defines the representation of an AADL instance model. The instance representation has been designed with several objectives:

- The ability to generate instance models from the declarative AADL model with a system implementation as the instance root.
- The ability to process an instance model without necessarily loading the complete declarative model. In other words, caching of property values that are relevant to a particular analysis or generation process in the instance model.
- The ability to represent a modal AADL instance model without necessarily creating a separate instance model for each mode combination, i.e., for each system operation mode as defined in section 11.
- The ability for tools that operate on the instance model to attach their results as properties of the instance model and have the property values be specific to each system operation mode.

The meta model for the AADL instance model is shown in the next three figures. Figure 35 shows the class hierarchy of instance model object classes. Figure 36 illustrates the relationships between AADL instance model objects. Figure 37 specifies classes that permit AADL instance model configuration information to be separately maintained.

The *InstanceObject* class is the common super class. It is defined as a subclass of PropertyHolder, i.e., all instance objects can contain property associations.

Different subclasses of *InstanceObject* have references to their respective classes in the declarative model (see Figure 35). For example, a *ComponentInstance* object refers to the subcomponent it is instantiated from. Instance objects other than *ConnectionInstance* objects refer to a single counterpart in the declarative model. A *ConnectionInstance* object refers to the collection of connection declarations that make up the connection instance. In addition, the *ConnectionInstance* object refers to the collection of *ComponentInstance* objects that provide the context of each of the connection declaration. The context of a connection declaration is the *ComponentInstance* object whose component implementation contains the connection declaration. This context is useful for modal processing of instance models.

The *ComponentInstance* class represents instances of subcomponents from the declarative AADL model. The subcomponent name is stored in the name attribute of the *ComponentInstance* and the category is recorded in the *category* attribute. The category attribute permits the component category to be determined without accessing the subcomponent in the declarative AADL model. The reference association to the subcomponent is a cross XML document reference, given that instance models and packages of AADL specifications are stored persistently in separate XML documents. Lazy loading of XML documents permits analysis and generation tools to operate on instance models without causing the declarative model to be loaded.

ComponentInstance objects are used to represent the full component instance hierarchy of a system. For example, a system instance model may consist of a *SystemInstance* object that contains a *ComponentInstance* for an application system and a *ComponentInstance* for a execution platform system. The application component instance contains *ComponentInstance* objects representing processes, which in turn contain *ComponentInstance* objects representing threads in those processes. Similarly, the execution platform component instance contains *ComponentInstance* objects representing processors, buses, memory, and devices.

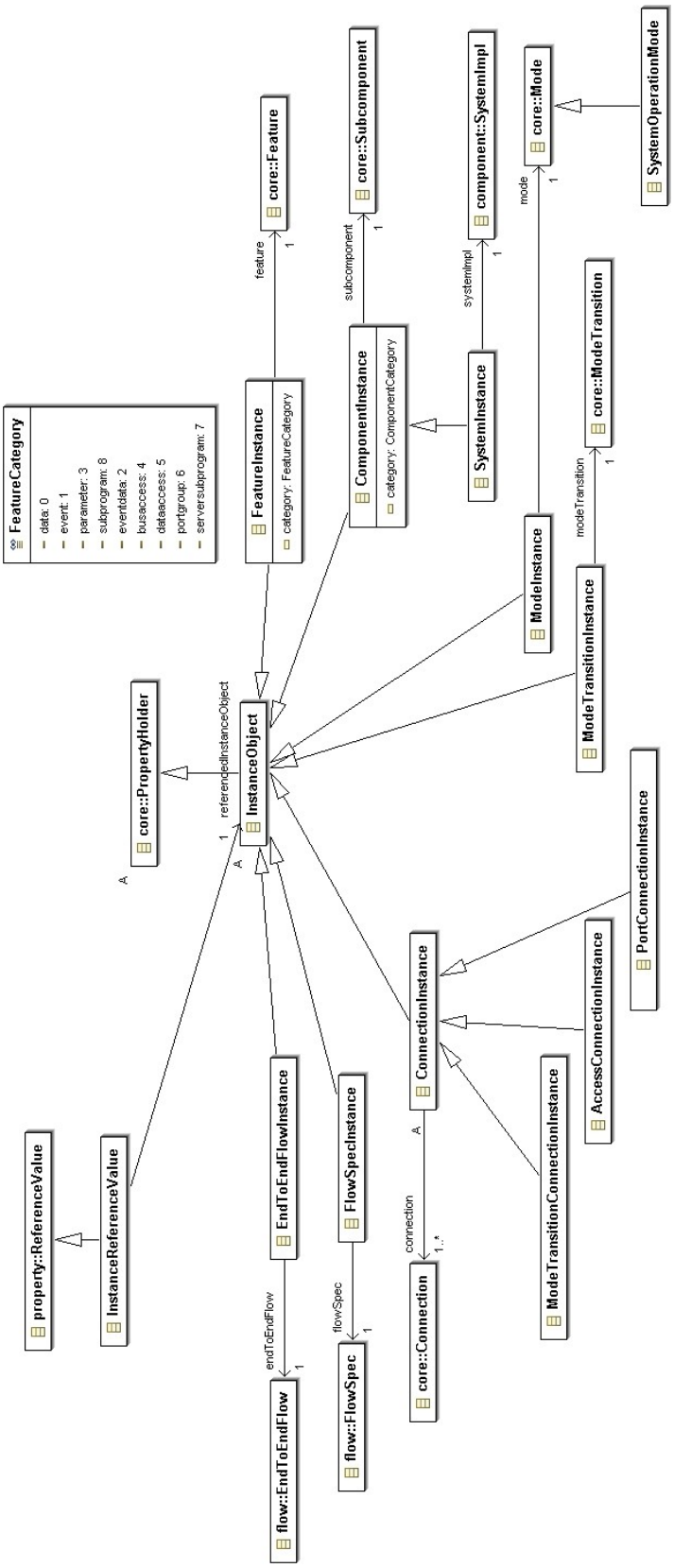

Figure 35 The AADL Instance Model Class Hierarchy

The definition of the *ComponentInstance* class in the AADL meta model permits a flattened instance model to be represented that is populated with only the leaf nodes of the instance hierarchy. This may be desirable if the intermediate components are not relevant for processing the instance model. In the example above, the instance model would consist of a *SystemInstance* object that contains *ComponentInstance* objects for all the threads, processors, buses, memory, and devices, but no intermediate components.

The *SystemInstance* class represents the instance root object that is the result of instantiating a system implementation. The *SystemInstance* has the constraint that it does not have a reference association to *Subcomponent* although it is a subclass of *ComponentInstance*. This reflects the fact that a *SystemInstance* is instantiated from a system implementation instead of a subcomponent.

Modal component instances, i.e., component instances whose subcomponent classifiers have mode declarations, contain objects of the *ModeInstance* class. Component instances also contain objects of the *FeatureInstance* class. The *FeatureInstance* class has a containment association to itself to support the representation of nested features in port groups. The feature name is stored in the name attribute of the *FeatureInstance* and the feature category is recorded in the *category* attribute.

The abstract *ConnectionInstance* class and its subclasses represent semantic connections. *PortConnectionInstance* objects represent semantic connections between ports and port groups. If the ultimate source or ultimate desintation of the semantic connection is a port group, then the semantic connection is unfolded into separate semantic connections for each of the event, data, and event data ports contained in the port group. *ModeTransitionConnectionInstance* objects represent semantic connections from event ports as ultimate sources to mode transitions. *AccessConnectionInstance* objects represent a semantic connection from the data or bus component to being accessed to a thread, device, memory, or processor requiring access to data or a bus.

It may be desirable to instantiate incomplete system models, i.e., system models where the application is not yet elaborated down to the thread level. For example, it may be desirable to perform flow related latency analysis on partitioned systems without the details of partition implementations. In this case, the component hierarchy is instantiated for subcomponents with classifiers and *ConnectionInstance* objects may be created between the leaf nodes of this incomplete system hierarchy. In this case, a *ConnectionInstance* object may not have a thread, processor, or device as its ultimate source or destination, thus, does not represent a semantic connection as defined in Section 9 of the AADL standard.

Flows are represented in AADL instance models as follows. The *FlowSpecInstance* class represents flow specifications associated with a component instance. The *EndToEndFlowInstance* class represents an end-to-end flow specification in the AADL instance model. They consist of a sequence of *FlowElementInstance* objects that alternate between a reference to a *FlowSpecInstance* object and a *PortConnectionInstance* object. *EndToEndFlowInstance* objects are contained in the component instance, whose component implementation contains the end-to-end flow declaration.

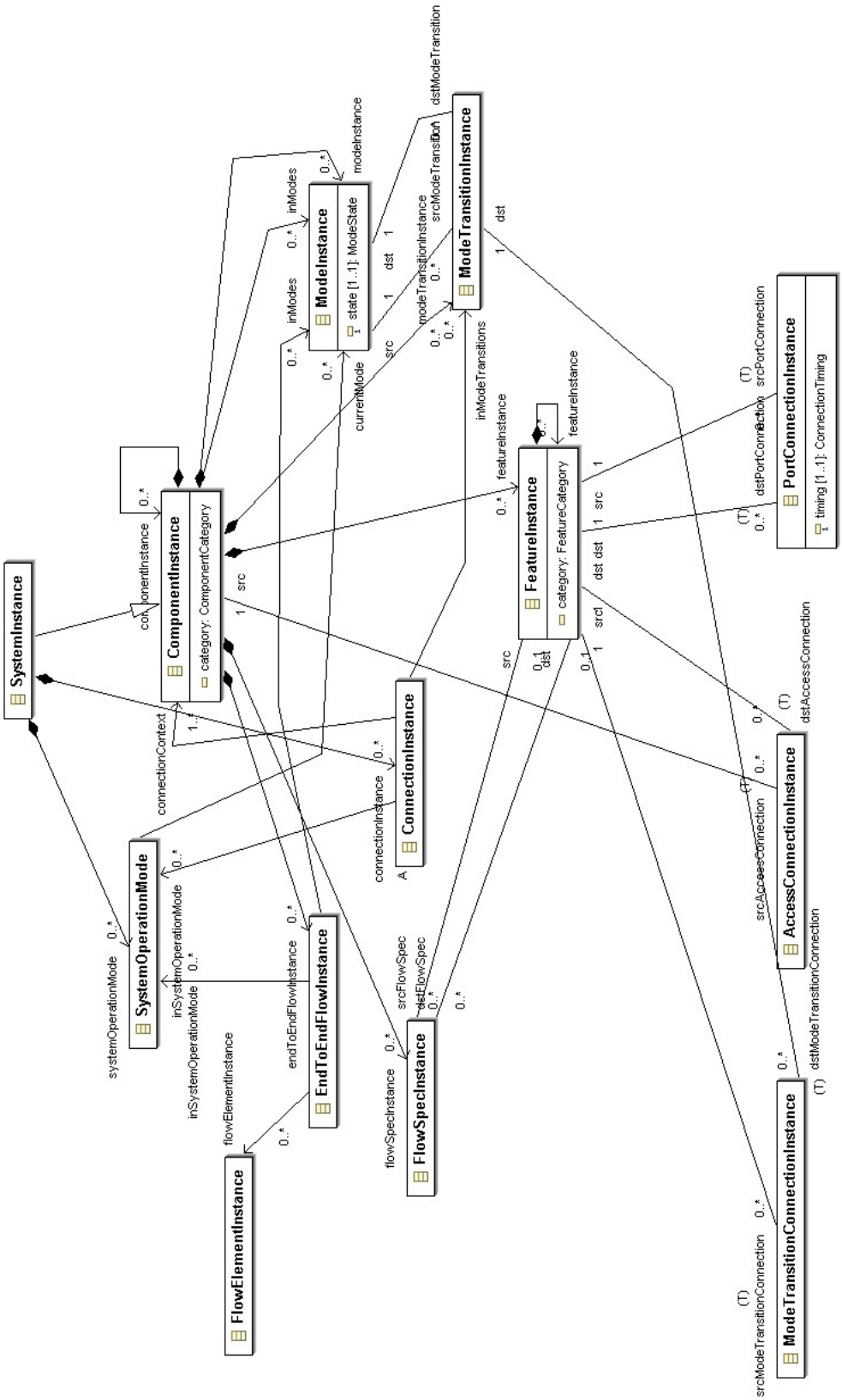

Figure 36 The AADL Instance Model Relationships

The paths declared by reference values of properties are expressed relative to the component for which the property association is declared. However, through the contained property association declaration the property value is actually associated with a component within the hierarchy below the component with the declaration. When property values are cached in the instance model they are attached to the instance object for which they are declared. Since all *InstanceObject* objects are property holders, the property association representation from the declarative AADL model can be used, except that the reference values have to be translated into references within the instance model. This is done through the *InstanceReferenceValue* class.

When property values are cached from the declarative model in the instance model, the property association has its *derived* attribute set to true. The property value of the property association is the value retrieved from the declarative model. A copy of the cached property value is attached to each instance object for which the property value holds. In other words, in the instance model the property value is found with the instance object and does not have to be searched for according to the rules for determining a property value as defined for property associations in the SAE AADL standard.

The *SystemOperationMode* class represents a system operation mode, i.e., a set of current modes of all active modal component instances. It is expressed as a multiplicity reference association to the relevant *ModeInstance* objects of modal component instances.

All valid system operation modes can be generated and attached to the system instance through a containment association. This allows analysis tools to iterate over the system operation modes to analyze the modal instance model.

Several *InstanceObject* classes represent model objects that are active in certain modes as expressed by the *in modes* clause. *ComponentInstance* objects are active in modes defined by the containing component instance. This is represented by the reference association *inModes* between *ComponentInstance* and *ModeInstance*. *ConnectionInstance* objects are active in combinations of modes of components whose connection declarations make up the connection instance. This mode combination is represented by *SystemOperationMode* objects. A reference association *inSystemOperationModes* between the *ConnectionInstance* and *SystemOperationMode* records this reference. *EndToEndFlowInstance* objects are active in modes defined by the containing component instance, which is recorded by an *inModes* reference association between *EndToEndFlowInstance* and *ModeInstance*. The system operation modes an end-to-end flow instance applies to is determined by these modes and the system operation modes that apply to each of its flow element instances. That set can be pre-calculated and recorded through the *inSystemOperationModes* reference association between *EndToEndFlowInstance* and *SystemOperationMode*.

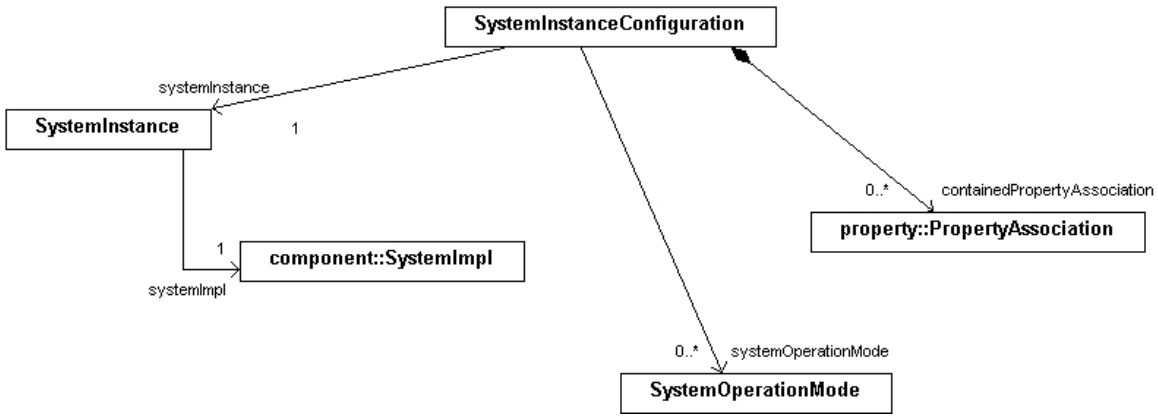

Figure 37 System Instance Configurations

A *SystemInstanceConfiguration* class has been defined to compactly represent different modal system configurations as a combination of a *SystemOperationMode* objects and property associations specific to those system operation modes. The property associations are contained property associations that apply to specific objects in the AADL instance model. This allows analysis tools to record their results in a more compact persistent form and can then be mapped onto the instance model. Multiple sets of of property values can be mapped to the same instance model.

Annex C.6 Graphical Layout Model

This representation is currently in development as part of the Graphical Layout Editor and will utilize an industry standard representation as appropriate.

