SAE INTERNATIONAL

AEROSPACE STANDARD

AS5506™ REV. D

Draft 2019-05

Superceding AS5506C

Architecture Analysis and Design Language (AADL)

RATIONALE

This Architecture Analysis & Design Language (AADL) standard document was prepared by the SAE AS-2C Architecture Description Language Subcommittee, Embedded Computing Systems Committee, Aerospace Avionics Systems Division.

The language was originally published as SAE AS5506 in 2004. The language has been refined and extended based on industrial experience as AADL V2 and published as AS5506A in 2009. The improvements focused on better support for architecture templates and modeling of layered and partitioned architectures. AADL V2.1 and V2.2, revisions that address a number of errata and minor improvements agreed upon by the committee, were published as AS5506B in 2012 and AS5506C in 2017.

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

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

TABLE OF CONTENTS

No table of figures entries found.

1. AADL STATIC SEMANTICS

Description

(1) An AADL specification is first governed by a set of static semantics rules that specify how model elements are organized in terms of containment hierarchy, allowed relationships (connections, bindings) with other components, XXX, allowed subclauses. Dynamic semantics provide an additional set of rules to interpret the behavior of atomic AADL components and how these behaviors can be composed to describe the execution of an AADL model.

(2) The purpose of this part is document the static semantics of component categories. Properties that can apply to component categories, dynamic semantics and consideration on modeling guidelines are presented in subsequent parts of this document.

(3) An AADL component can have one of several categories:

(4) Software category refers to software elements as they are regularly seen in computer systems: `thread`, `thread group`, `process`, `data`, `subprogram`, `subprogram group`;

(5) Execution platform category refers to actual elements that support the execution of software: `memory`, `virtual memory`, `processor`, `virtual processor`, `bus`, `virtual bus`, `device`;

(6) Generic category refers to `abstract` components whose actual semantics is undefined, and would be later refined;

(7) Composite category refers to `system` components as a unit of composition. Components are combined as a collection of interconnected subcomponents.

2. SOFTWARE COMPONENTS

(8) This section defines the following categories of software components: data, subprogram, subprogram group, thread, thread group, and process.

(9) Software components may have associated source text specified using property associations. Software source text can be processed by source text tools to obtain a binary executable image consisting of code and data to be loaded onto a memory component and executed by a processor component. Source text may be written in a traditional programming language, a very-high-level or domain-specific language, or may be an intermediate product of processing such representations, e.g., an object file.

(10) Data components classifiers represent data types, while data subcomponents represent static data in source text, and local variables in subprograms. Data components are sharable between threads within the same thread group or process, and across processes and systems.

(11) The subprogram component models callable source text that is executed sequentially. Subprograms are callable from within threads and subprograms.

(12) Threads represent sequential sequences of instructions in loaded binary images produced from source text. Threads model schedulable units of control that can execute concurrently. Threads can interact with each other through exchanges of control and data as specified by port connections, through remote subprogram calls, and through shared data components.

(13) A thread group is a compositional component that permits organization of threads within processes into groups with relevant property associations.

(14) A process represents a virtual address space. Access protection of the virtual address space is by default enforced at runtime, but can be disabled if specified by the property `Runtime_Protection`. The source text associated with a process forms a complete program as defined in the applicable programming language standard. ~~A complete process specification must contain at least one thread declaration.~~ Processes may share a data component as specified by the required subcomponent resolved to an actual subcomponent and accessed through port connections.

2.1 Subprograms ~~and Subprogram Calls~~

(15) A subprogram component represents sequentially executed source text that is called with parameters.

(16) A subprogram may not have any state that persists beyond the call (static data). Subprograms can have local variables that are represented by data subcomponents in the subprogram implementation. All parameters and required access to persistent data must be explicitly declared as part of the subprogram type declaration. In addition, any events raised within a subprogram must be specified as part of its type declaration.

(17) Subprograms can be called from threads and from other subprograms that execute within a thread. These calls can be local calls, i.e., performed in the context of the caller thread, or they can be remote calls to subprograms that are executed in the context of another thread.

(18) Subprogram instances may be modeled explicitly through subprogram subcomponent declarations. In this case, components can be modeled as requiring and providing access to subprogram instances.

(19) Subprogram instances may be implied by a reference to a subprogram as part of property value, or through the use of annex language. In this case, a subprogram is implicitly instantiated within the containing process.

Legality Rules

Category	Type	Implementation
subprogram	Features: • out ~~event~~ port • ~~out event data port~~ • feature group • requires data access • requires subprogram access • requires subprogram group access • parameter • feature Flow specifications: yes Modes: yes Properties: yes	Subcomponents: • data • abstract • subprogram Connections: yes Flows: yes Modes: yes Properties: yes

(L1) A subprogram type declaration can contain parameter, out ~~event port, out event data~~ port, and feature group declarations as well as requires data, subprogram, and subprogram group access declarations. It can also contain a flow specification subclause, a modes subclause, and property associations.

(L2) A subprogram implementation can contain abstract, subprogram, and data subcomponents, a connections subclause, a flows subclause, a modes subclause, and property associations.

Static Semantics

(20) A subprogram component represents sequentially executable source text that is called with parameters. The results of a subprogram call must be available to the caller at the time those results are used. This allows for synchronous and semi-synchronous calls.

(21) A subprogram type declaration specifies all interactions of the subprogram with other parts of the application source text. Subprogram parameters are specified as features of a subprogram type. This includes **in** and **in out** parameters passed into a subprogram and **out** and **in out** parameters returned from a subprogram on a call, events being raised from within the subprogram through its **out event port** and **out event data port**. Required access to static data by the subprogram are specified as part of the features subclause of a subprogram type declaration, and required access to subprograms that are contained in another component and are called by this subprogram.

(22) A subprogram implementation represents implementation details that are relevant to architecture modeling. It specifies calls to other subprograms and the mode in which the call sequence occurs. It also specifies any local data of the subprogram~~, i.e., data that does not persist beyond the call~~.

(23) Detailed modeling of subprograms is not prescribed. The level of detail is determined by the constructs necessary for performing architecture analyses or code generation.

(24) All access to data that persists beyond the life of the subprogram execution, i.e., any state that is maintained by a subprogram, must be modeled through requires data access. If requires data access is declared for a subprogram type, access to the data subcomponent may be performed in a critical region to assure concurrency control for calls from different threads (for more on concurrency control see Sections XXX).

(25) Subprogram implementations and thread implementations can contain subprograms as subcomponents. The corresponding subcomponents, connected to the enclosing components features or other sibling components form a call graph, or a graph of subprogram calls, with parameters flowing through each node of the call graph using actual AADL connection semantics for features. A method of implementation can decide to turn this flow graph into a sequence of subprogram calls, or treat them as a parallel execution of subprograms.

(26) A subprogram subcomponent declaration explicitly represents a subprogram instance that resides in the protected address space of the process that hosts its execution.

(27) If a subprogram is used multiple time inside the same memory space, a method of implementation may decide to duplicate the corresponding code, or reuse it across multiple usage domains. This allows for time/memory optimization strategies.

(28) In the case of remote subprogram calls a proxy may be loaded for the calling thread and the actual subprogram is part of the load image of the process with the thread servicing the remote subprogram call.

(29) In an object-oriented application methods are called on an object instance and the object instance is available within the method by the name *this*. In AADL a subprogram call can identify the subprogram being called by the provides subprogram access feature of a data component. In AADL, the data component must be explicitly passed into a subprogram as parameter (by value) or as requires data access (by reference). Requires data access may require concurrency control to ensure mutual exclusion.

Examples

TBD

2.2 Subprogram Groups and Subprogram Group Types

(1) Subprogram groups represent subprogram libraries.

(2) Subprogram groups can be made accessible to other components through subprogram group access features (see Section XXX) and subprogram group access connections. This grouping concept allows the number of connection declarations to be reduced, especially at higher levels of a system when a number of provided subprograms from one subcomponent and its contained subcomponents must be connected to requires subprogram access in another subcomponent and its contained subcomponents. The content of a subprogram group is declared through a subprogram group type declaration. This declaration is then referenced when subprogram groups are declared as subcomponents.

Naming Rules

~~(N1) The defining identifier of a subprogram group type must be unique within the package namespace of the package where the subprogram group type is declared.~~

~~(N2) Each subprogram group provides a local namespace. The defining subprogram identifiers of subprogram declarations in a subprogram group type must be unique within the namespace of the subprogram group type.~~

~~(N3) The local namespace of a subprogram group type extension includes the defining identifiers in the local namespace of the subprogram group type being extended. This means, the defining identifiers of subprogram or subprogram group declarations in a subprogram group type extension must not exist in the local namespace of the subprogram group type being extended. The defining identifiers of subprogram or subprogram group refinements in a subprogram group type extension must refer to a subprogram or subprogram group in the local namespace of an ancestor subprogram group type.~~

~~(N4) The defining subprogram identifiers of subprogram access feature declarations in feature group refinements must not exist in the local namespace of any subprogram group being extended. The defining subprogram identifier of `subprogram_refinement` declarations in subprogram group refinements must exist in the local namespace of any feature group being extended.~~

~~(N5) The package name of the unique subprogram group type reference must refer to a package name in the global namespace. The subprogram group type identifier of the unique subprogram group type reference must refer to a subprogram group type identifier in the named package.~~

Legality Rules

Category	Type	Implementation
subprogram group	Features: • feature group • provides subprogram access • requires subprogram access • requires subprogram group access • provides subprogram group access • feature Flow specifications: no Modes: no Properties: yes	Subcomponents: • subprogram • subprogram group • data • abstract Connections: yes Flows: no Modes: no Properties: yes

(L3) A subprogram group type can contain provides and requires subprogram access, and provides and requires subprogram group access.

(L4) A subprogram group implementation can contain abstract, data, subprogram group, and subprogram subcomponents as well as data and subprogram access connections.

(L5) A subprogram group type or implementation may contain zero or more subcomponent declarations. If it contains zero elements, then the subprogram group type or implementation is considered to be incompletely specified.

Semantics

(30) A subprogram group declaration represents groups of component subprograms, i.e., subprogram libraries. Subprograms in a subprogram group may require access to other subprograms or subprogram groups.

~~(31) Requires subprogram group access is resolved to provides subprogram group access or a subprogram group subcomponent.~~

~~(32) The subprograms of a subprogram group or a subprogram group access feature can be connected to or referenced in a subprogram call.~~

Processing Requirements and Permissions

(33) Subprogram groups represent subprogram libraries. As a consequence, these can be application libraries or system libraries, and may be shared across multiple applications, i.e., across multiple processes.

(34) Methods of implementation may optionally allow a provides subprogram access declaration of a subprogram group to not be connected to a subprogram instantiation, i.e., subprogram subcomponent. It may assume these subprograms to be implicitly declared and instantiated as part of a subprogram group instantiation. E.g. when the corresponding library is loaded in memory.

Examples

```
subprogram group mathlib
features
   matrixMultiply: provides subprogram access ;
```

```
   matrixAdd: provides subprogram access ;
   vectorAdd: requires subprogram access ;
end mathlib;
```

2.3 Threads

(1) A thread models a concurrent task or an active object, i.e., a schedulable unit that can execute concurrently with other threads. Each thread represents a sequential flow of control that executes instructions within a binary image produced from source text.

(2) AADL supports an input-compute-output model of communication and execution for threads and port-based communication. The inputs received from other components are frozen at a specified point, by default the dispatch of a thread. As a result the computation performed by a thread is not affected by the arrival of new input until an explicit request for input, by default the next dispatch. Similarly, the output is made available to other components at a specified point in time, for data ports by default at completion time or thread deadline. In other words, AADL is able to support both synchronous execution and communication behavior, e.g., in the form of deterministic sampling of a control system data stream, as well as asynchronous concurrent processing.

(3) Systems modeled in AADL can have operational modes (see Section XX). A thread can be active in a particular mode and inactive in another mode. As a result, a thread may transition between an active and inactive state as part of a mode switch. Only active threads can be dispatched and scheduled for execution. Threads can be dispatched periodically or as the result of explicitly modeled events that arrive at event ports, event data ports. Completion of the normal execution including error recovery will result in an event being delivered through the reserved `Complete` event out port. Completion under unrecoverable error conditions will result in an event being delivered through the reserved `Abort` and `Stop` ports.

(4) If the thread execution results in a fault that is detected, the source text may handle the error. If the error is not handled in the source text, the thread is requested to recover and prepare for the next dispatch. If an error is considered thread unrecoverable, its occurrence is reported through the reserved `Error` out event data port.

Legality Rules

Category	Type	Implementation
thread	Features: • port • feature group • provides data access • requires data access • provides subprogram access • requires subprogram access • provides subprogram group access • requires subprogram group access • feature Flow specifications: yes Modes: yes Properties: yes	Subcomponents: • data • subprogram • subprogram group • abstract Subprogram calls: yes Connections: yes Flows: yes Modes: yes Properties: yes

(L6) A thread type declaration can contain port, feature group, requires and provides data access declarations, as well as requires and provides subprogram access declarations. It can also contain flow specifications, a modes subclause, and property associations.

(L7) A thread component implementation can contain abstract, data, subprogram, and subprogram group subcomponent declarations, a calls subclause, a flows subclause, a modes subclause, and thread property associations.

(L8) The `Complete` **out** event port, `Error` **out** event data port, `Abort` **out** event port, and `Stop` **out** event port are reserved, i.e., they must be declared of the specified port type. They must be explicitly defined as features in order to be referenced, e.g., in connections, flows and mode transitions.

Examples

```
thread Prime_Reporter
features
  Received_Prime : in event data port Base_Types::Integer;
properties
  Dispatch_Protocol => Timed;
end Prime_Reporter;

thread Prime_Reporter_One extends Prime_Reporter
features
  Received_Prime : refined to in event data port Base_Types::Integer
    {Compute_Entrypoint_Source_Text => "Primes.On_Received_Prime_One";};
    -- function called on message-based dispatch
properties
  Period => 9 Sec; -- timeout period
  Priority => 45;
  Compute_Entrypoint_Source_Text => "Primes.Report_One";
  -- function called in case of timeout
end Prime_Reporter_One;
```

2.4 Thread Groups

(1) A thread group represents an organizational component to logically group threads contained in processes. The type of a thread group component specifies the features and required subcomponent access through which threads contained in a thread group interact with components outside the thread group. Thread group implementations represent the contained threads and their connectivity. Thread groups can have multiple modes, each representing a possibly different configuration of subcomponents, their connections, and mode-specific property associations. Thread groups can be hierarchically nested.

(2) A thread group does not represent a virtual address space nor does it represent a unit of execution. Therefore, a thread group must be directly or indirectly contained within a process.

Legality Rules

Category	Type	Implementation
thread group	Features: • port • feature group • provides data access • requires data access • provides subprogram access • requires subprogram access • provides subprogram group access • requires subprogram group access • feature Flow specifications: yes Modes: yes Properties: yes	Subcomponents: • data • subprogram • subprogram group • thread • thread group • abstract Subprogram calls: no Connections: yes Flows: yes Modes: yes Properties: yes

(L9) A thread group component type can contain provides and requires data access, as well as port, feature group, provides and requires subprogram access declarations, and provides and requires subprogram group access declarations. It can also contain flow specifications, modes subclauses, and property associations.

(L10) A thread group component implementation can contain abstract, data, subprogram, subprogram group, thread, and thread group subcomponent declarations.

(L11) A thread group implementation can contain a connections subclause, a flows subclause, a modes subclause, and properties subclause.

(L12) A thread group must not contain a subprogram calls subclause.

Standard Properties

-- Properties related to source text

Source_Text: **inherit list of aadlstring**

-- Inheritable thread properties

Synchronized Component: **inherit aadlboolean => true**

Active Thread Handling Protocol:
 inherit Supported Active Thread Handling Protocols => abort
Period: **inherit** Time

Deadline: **inherit** Time => Period

Dispatch Offset: **inherit** Time

First Dispatch Time : **inherit** Time

-- Scheduling properties

Priority: **inherit aadlinteger**

Time Slot: **list of aadlinteger**

Criticality: **aadlinteger**

-- execution time related properties

Reference Processor: **inherit classifier** (processor)

-- mode related properties

Resumption Policy: **enumeration** (restart, resume)

-- startup properties

Startup Deadline: Time

Startup Execution Time: Time Range

-- Properties specifying constraints for processor and memory binding

Allowed Processor Binding Class:
 inherit list of classifier (processor, virtual processor, device, system)

Allowed Processor Binding: **inherit list of reference** (processor, virtual processor, device, system)

Allowed Memory Binding Class:
 inherit list of classifier (memory, system, processor, virtual processor)

Allowed Memory Binding: **inherit list of reference** (memory, system, processor, virtual processor)

Actual Processor Binding: **inherit list of reference** (processor, virtual processor, device, system)

~~Actual Memory Binding: **inherit list of reference** (memory, system, processor, virtual processor)~~

~~Allowed Connection Binding Class:~~
~~**inherit list of classifier**(processor, virtual processor, bus, virtual bus, device, memory, system)~~

~~Allowed Connection Binding: **inherit list of reference** (processor, virtual processor, bus, virtual bus, device, memory, system)~~

~~Actual Connection Binding: **inherit list of reference** (processor, virtual processor, bus, virtual bus, device, system, memory)~~

~~NOTE: Property associations of thread groups are inheritable (see Section 11.3) by contained subcomponents. This means if a contained thread does not have a property value defined for a particular property, then the corresponding property value for the thread group is used.~~

Semantics

(3) A thread group allows threads contained in processes to be logically organized into a hierarchy. A thread group type declares the features and required subcomponent access through which threads contained in a thread group can interact with components declared outside the thread group.

(4) Thread groups may contain subprogram subcomponents and subprogram groups. The code of such subprograms and subprogram groups resides in the address space of the containing process. The subprograms may be called by threads contained in the thread group. The subprograms may also be called from outside the thread group if made accessible through a provides subprogram access declaration or subprogram group access declaration.

(5) Thread groups may contain data components. They represent state that may be shared between threads inside the thread group through access connections to the requires data access features of those threads, and shared outside the thread group through provides data access features of the thread group.

(6) A thread group implementation contains threads and thread groups. Thread group nesting permits threads to be organized hierarchically. A thread group implementation also contains connections to specify the interactions between the contained subcomponents and modes to represent different configurations of subsets of subcomponents and connections as well as mode-specific property associations.

2.5 Processes

(1) A process represents a virtual address space, i.e., it represents a space partition unit whose boundaries are enforced at runtime. ~~The Runtime_Protection process property indicates whether runtime protection is disabled.~~ The virtual address space contains the program formed by the source text associated with the process and its subcomponents. Threads of a process must be explicitly declared.

Legality Rules

Category	Type	Implementation
process	Features: • port • feature group • provides data access • requires data access • provides subprogram access • requires subprogram access • provides subprogram group access • requires subprogram group access • feature Flow specifications: yes Modes: yes Properties: yes	Subcomponents: • data • subprogram • subprogram group • thread • thread group • abstract Subprogram calls: no Connections: yes Flows: yes Modes: yes Properties: yes

(L13) A process component type can contain port, feature group, provides and requires data access, provides and requires subprogram access declarations, and provides and requires subprogram group access declarations. It can also contain flow specifications, modes subclause, and property associations.

(L14) A process component implementation can contain abstract, data, subprogram, subprogram group, thread, and thread group subcomponent declarations.

(L15) A process implementation can contain a connections subclause, a flows subclause, a modes subclause, and a properties subclause.

(L16) A thread group must not contain a subprogram calls subclause.

Consistency Rules

(C1) The complete source text associated with a process component must form a complete and legal program as defined in the applicable source language standard. This source text shall include the source text that corresponds to the complete set of subcomponents in the process's containment hierarchy along with the data and subprograms that are referenced by required subcomponent declarations.

~~Standard Properties~~

```
-- Runtime enforcement of virtual address space boundary
Runtime_Protection : inherit aadlboolean
-- Properties related to source text
Source_Text: inherit list of aadlstring
Source_Language: inherit list of Supported_Source_Languages
-- Properties related to virtual address space loading
Load_Time: Time_Range
Load_Deadline: Time
-- Inheritable thread properties
Synchronized_Component: inherit aadlboolean => true

Active_Thread_Handling_Protocol:
   inherit Supported_Active_Thread_Handling_Protocols => abort
Period: inherit Time
Deadline: inherit Time => Period
```

~~Dispatch Offset: **inherit** Time~~

~~-- execution time related properties~~

~~Reference Processor: **inherit classifier** (processor)~~

~~-- Scheduling related properties~~

~~Priority: **inherit aadlinteger**~~

~~-- mode related properties~~

~~Resumption Policy: **enumeration** (restart, resume)~~

~~Deactivation Policy: **enumeration** (inactive, unload) -> inactive~~

~~-- process initialization~~

~~Startup Deadline: Time~~

~~Startup Execution Time: Time Range~~

~~-- Properties specifying constraints memory binding~~

~~Allowed Processor Binding_Class:~~
~~**inherit list of classifier** (processor, virtual processor, device, system)~~

~~Allowed Processor Binding: **inherit list of reference** (processor, virtual processor, device, system)~~

~~Actual Processor Binding: **inherit list of reference** (processor, virtual processor, device, system)~~

~~Allowed Connection Binding Class:~~
~~**inherit list of classifier**(processor, virtual processor, bus, virtual bus, device, memory, system)~~

~~Allowed Connection Binding: **inherit list of reference** (processor, virtual processor, bus, virtual bus, device, memory, system)~~

~~Actual Connection Binding: **inherit list of reference** (processor, virtual processor, bus, virtual bus, device, system, memory)~~

~~Allowed Memory Binding Class:~~
~~**inherit list of classifier** (memory, system, processor, virtual processor)~~

~~Allowed Memory Binding: **inherit list of reference** (memory, system, processor, virtual processor)~~

~~Actual Memory Binding: **inherit list of reference** (memory, system, processor, virtual processor)~~

Semantics

(2) Every process has its own virtual address space. This address space provides access to source code and data associated with the process and all its contained components. This address space boundary is by default enforced at runtime, but can be disabled through the `Runtime_Protection` property.

(3) Threads contained in a process execute within the virtual address space of the process.

(4) Processes may contain subprogram subcomponents. The code of such subprograms resides in the address space of the process. The calling semantics to such subprograms are defined in Section XX.

(5) A process may contain mode declarations. In this case, each mode can represent a different configuration of contained threads, their connections, and mode-specific property associations. The transition between modes is determined by the mode transition declarations and is triggered by the arrival of *mode transition trigger events* (see Sections 12 and 13.6).

(6) The associated source text for each process is compiled and linked to form binary images in accordance with the applicable source language standard. These binary images must be loaded into memory before any thread contained in a process can execute, i.e., enter its *perform thread initialization* state.

(7) The time to load binary images into the virtual address space of a process is bounded by the `Load_Deadline` and `Load_Time` properties. The failure to meet these timing requirements is considered an error.

3. EXECUTION PLATFORM COMPONENTS

(8) This section describes the categories of execution platform components that represent computing hardware: processor, virtual processor, memory, bus, virtual bus; and the physical environment: device.

(9) Processors can execute threads. Processors can contain memory subcomponents. Processors can access memories and communicate with devices and other processors over buses. Threads, thread groups, and processes are bound to processors.

(10) Virtual processors are logical execution platform elements that can execute threads. Threads, thread groups, and processes can be bound to virtual processors. Virtual processors must be bound to or contained in processors. This determines the binding of threads to processors.

(11) Memories represent randomly addressable storage capable of storing binary images in the form of data and code. Memories can be accessed by executing threads.

(12) Buses support physical communication between processors, devices, and memories. A bus provides the resources necessary to perform exchanges of control and data as specified by connections. These resources include bandwidth and protocols to perform the exchange. A connection may be bound to a sequence of buses and intermediate processors and devices in a manner that is analogous to the binding of threads to processors.

(13) Virtual buses represent a logical bus abstraction to model protocols and virtual channels. Virtual buses can be contained in or bound to processors and buses. Connections can specify that they require specific protocols, or certain quality of service from protocols of platform components they are bound to.

(14) Devices represent entities of the physical environment, e.g., an engine, or entities that interface with the physical environment, e.g., a sensor or an actuator. A device can interact with application software components through their port and provides subprogram access features. A device may interact with other execution platform components through bus access connections. A device may achieve its functionality through device internal software or may require device driver software to be executed by a processor. Binary images or threads cannot be bound to devices.

(15) Processors may include software that implements the capability of the processor to schedule and execute threads and other services of the processor. Its source text and data in the form of binary images will be bound to memories accessible from that processor. The resource requirements of this software are reflected in processor properties.

(16) Execution platform components can be assembled into execution platform systems, i.e., into systems of execution platform components to model complex physical computing hardware components and software/hardware computing systems, through the use of system components (see Section 7.1). The execution platform systems and their components may denote physical computing hardware for example, memory to represent a hard disk or RAM. Execution platform systems may also model abstracted storage, for example, a device or memory to represent a database, depending on the purpose of the model.

(17) The hardware represented by the execution platform components may be modeled in a hardware description or simulation language. Alternatively, it may be represented using configuration data for programmable logic devices. A simulation may be used to characterize the components. Such descriptions can be associated with the component by property association.

(18) Execution platform components can represent high-level abstractions of physical and computing components. A detailed AADL model of their implementation can be represented by system implementations that are associated with the execution platform component by property (see Section 14). This effectively models a layered system architecture.

3.1 Processors

(19) A processor is an abstraction of hardware and software that is responsible for scheduling and executing threads and virtual processors that are bound to it. A processor also may execute driver software that is declared as part of devices that can be accessed from that processor. Processors may contain memories and may access memories and devices via buses.

Legality Rules

Category	Type	Implementation
processor	Features: • provides subprogram access • provides subprogram group access • port • feature group • requires bus access • provides bus access • requires virtual bus access • provides virtual bus access • feature Flow specifications: yes Modes: yes Properties: yes	Subcomponents: • memory • bus • virtual processor • virtual bus • abstract Subprogram calls: no Connections: yes Flows: yes Modes: yes Properties: yes

(L17) A processor component type can contain port, feature group, provides subprogram access, provides subprogram group access, virtual buss access, and bus access declarations. It may contain flow specifications, a modes subclause, as well as property associations.

(L18) A processor component implementation can contain declarations of memory, bus, virtual bus, virtual processor, and abstract subcomponents.

(L19) A processor implementation can contain a modes subclause, flows subclause, and a properties subclause.

(L20) A processor implementation can contain bus access, subprogram access, subprogram group access, port, feature, and feature group connection declarations.

(L21) A processor implementation must not contain a subprogram calls subclause.

~~Standard Properties~~

~~-- Hardware description properties~~

~~Hardware Description Source Text: **inherit list of aadlstring**~~

~~Hardware Source Language: Supported Hardware Source Languages~~

~~-- Properties related to source text that provides thread scheduling services~~

~~Source Text: **inherit list of aadlstring**~~

~~Source Language: **inherit list of** Supported_Source_Languages~~

~~Code Size: Size~~

~~Data Size: Size~~

~~Stack Size: Size~~

~~Allowed Memory Binding Class:~~
~~**inherit list of classifier** (memory, system, processor, virtual processor)~~

~~Allowed Memory Binding: **inherit list of reference** (memory, system, processor, virtual processor)~~

~~Allowed Memory Binding: **inherit list of reference** (memory, system, processor, virtual processor)~~

~~-- Processor initialization properties~~

~~Startup Deadline: Time~~

~~Startup Execution Time: Time Range~~

~~-- Properties specifying provided thread execution support~~

~~Thread Limit: **aadlinteger** 0 .. Max Thread Limit~~

~~Allowed Dispatch Protocol: **list of** Supported Dispatch Protocols~~

~~Allowed Period: **list of** Time Range~~

~~Scheduling Protocol: **inherit list of** Supported Scheduling Protocols~~

~~Scheduler Quantum : **inherit** Time~~

~~Slot Time: Time~~

~~Frame Period: Time~~

~~-- acceptable priority value on threads and mapping into RTOS priority values~~

~~Priority Range: **range of aadlinteger**~~

~~Priority Map: **list of** Priority Mapping~~

~~Process Swap Execution Time: Time Range~~

~~Thread Swap Execution Time: Time Range~~

~~Supported Source Language: **list of** Supported Source Languages~~

~~-- Scaling of processor speed~~

~~Processor Capacity: **aadlreal** Processor Speed Units~~

~~Reference Processor: **inherit classifier** (processor)~~

~~-- Properties related to data movement in memory~~

~~Assign Time: **record** (~~

~~Fixed: Time Range;~~

~~PerByte: Time Range;)~~

~~-- Properties related to the hardware clock~~

~~Reference Time: **inherit reference** (processor, device, bus, system, abstract)~~

~~Clock Jitter: Time~~

~~Clock Period: Time~~

~~Clock Period Range: Time Range~~

~~-- Protocol support~~

~~Provided Virtual Bus Class : **inherit list of classifier** (virtual bus)~~

~~Provided Connection Quality Of Service : **inherit list of** Supported Connection QoS~~

~~-- mode related properties~~

~~Resumption Policy: **enumeration** (restart, resume)~~

~~Deactivation Policy: **enumeration** (inactive, unload) => inactive~~

~~-- runtime protection support of address spaces~~

~~Runtime Protection Support : **aadlboolean**~~

~~-- Virtual machine layering~~

~~Implemented As: **classifier** (system implementation, abstract implementation)~~

Semantics

(20) A processor is the execution platform component that is capable of scheduling and executing threads. Threads will be bound to a processor for their execution that supports the dispatch protocol required by the thread. The `Allowed_Dispatch_Protocol` property specifies the dispatch protocols that a processor supplies.

(21) Support for thread execution may be embedded in the processor hardware or it may require software that implements processor functionality such as thread scheduling, e.g., an operating system kernel or other software virtual machine. Such software must be bound to a memory component that is accessible to the processor via the `Actual_Memory_Binding` property. Services provided by such software can be called through the provides subprogram access features of a processor.

(22) The code that threads execute and the data they access must be bound to a memory component that is accessible to the processor on which the thread executes.

(23) If a processor executes device driver software associated with a device, then the processor must have access to the corresponding device component.

(24) Flow specifications model logical flow through processors. For example, they may represent requests for operating system services through subprograms or ports.

(25) The hardware description source text property may provide a reference to source text that is a model of the hardware in a hardware description language. This provides support for the simulation of hardware.

(26) Modes allow processor components to have different property values under different operational processor modes. Modes may be used to specify different runtime selectable configurations of processor implementations.

(27) A processor may have ports through which it reports information to the application software, e.g., to report error conditions. Those ports are identified in port connection declarations by the reserved word **processor** (see Section 9.1).

(28) A processor may have provide subprogram or subprogram group access to represent services that can be called by the application. A subprogram call identifies such a service by the subprogram classifier and the binding to the specific processor is implicit through the actual processor binding of the thread that contains the call.

(29) A processor component can include protocols in its abstraction. These protocols can be explicitly modeled as virtual bus subcomponents to satisfy protocol requirements by connections. The fact that a protocol is supported by a processor is specified by a `Provided_Virtual_Bus_Class` property.

(30) A processor can contain a bus subcomponent that it makes externally accessible. This models a bus that is managed by the processor and other components can connect to it. In this case the processor is implicitly connected to this bus.

(31) Different processors may be different execution speeds. This affects the execution time specified for threads and subprograms. The **in binding** statement of property associations permits processor type specific execution times to be declared. The execution time of a thread or subprogram can also be scaled according to scaling factors between different processors. The `Processor_Capacity property` specifies the speed of a processor for resource budget analysis. This property is also used to determine a scaling factor for execution time with respect to the processor capacity of a specified `Reference_Processor`.

(32) The processor is an abstraction of a hardware processor and possibly a runtime system. If it is desirable, the internals of the processor can be modeled by AADL as a system in its own right, i.e., an application system and an execution platform. This system implementation description can be associated with the processor component declaration by the `Implemented_As` property (see Section 14).

3.2 Virtual Processors

(33) A virtual processor represents a logical resource that is capable of scheduling and executing threads and other virtual processors bound to them. Virtual processors can be declared as a subcomponent of a processor or another virtual processor, i.e., they are implicitly bound to the processor or virtual processor they are contained in. Virtual processors can also be declared separately, that is as a subcomponent of a system component, and explicitly bound to a processor or virtual processor. This allows virtual processors to represent hierarchical schedulers that schedule thread and/or virtual processors bound to them. In a fully bound system every virtual processor and thread is eventually bound to a physical processor. Virtual processors can be interconnected via virtual buses. This allows users to model virtual platforms.

Legality Rules

Category	Type	Implementation
virtual processor	Features: • provides subprogram access • provides subprogram group access • port • requires virtual bus access • provides virtual bus access • feature group • feature Flow specifications: yes Modes: yes Properties: yes	Subcomponents: • virtual processor • virtual bus • abstract Subprogram calls: no Connections: yes Flows: yes Modes: yes Properties: yes

(L22) A virtual processor component type can contain port, feature group, provides subprogram access, and subprogram group access declarations. It may contain flow specifications, a modes subclause, as well as property associations.

(L23) A virtual processor component implementation can contain declarations of virtual bus, virtual processor, and abstract subcomponents.

(L24) A virtual processor implementation can contain a modes subclause, flows subclause, and a properties subclause.

(L25) A virtual processor implementation must not contain a subprogram calls subclause.

(L26) A virtual processor implementation can contain subprogram access, subprogram group access, port, feature, and feature group connections.

Consistency Rules

(C2) In a fully bound system every virtual processor must be directly or indirectly bound to, or directly or indirectly contained in a physical processor.

(C3) There must not be cycles in bindings between virtual processors.

Standard Properties

-- Properties related to source text that provides thread scheduling services

Source Text: **inherit list of aadlstring**

Source Language: **inherit list of** Supported Source Languages

Code Size: Size

Data Size: Size

Stack Size: Size

Allowed Memory Binding Class:
inherit list of classifier (memory, system, processor, virtual processor)

Allowed Memory Binding: **inherit list of reference** (memory, system, processor, virtual processor)

Actual Memory Binding: **inherit list of reference** (memory, system, processor, virtual processor)

Allowed Processor Binding Class:
inherit list of classifier (processor, virtual processor, device, system)

Allowed Processor Binding: **inherit list of reference** (processor, virtual processor, device, system)

Actual Processor Binding: **inherit list of reference** (processor, virtual processor, device, system)

-- Virtual processor initialization properties

Startup Execution Time: Time Range

Startup Deadline: Time

-- Properties specifying provided thread execution support

Thread Limit: **aadlinteger** 0 .. Max Thread Limit

Allowed Dispatch Protocol: **list of** Supported Dispatch Protocols

Allowed Period: **list of** Time Range

Scheduling Protocol: **inherit list of** Supported Scheduling Protocols

Slot Time: Time

Frame Period: Time

-- acceptable priority value on threads and mapping into RTOS priority values

Priority Range: **range of aadlinteger**

Priority Map: **list of** Priority_Mapping

Process Swap_Execution Time: Time Range

Thread Swap Execution Time: Time Range

Supported Source Language: **list of** Supported Source Languages

-- Properties of the dispatch characterstics of this virtual processor

Period: **inherit** Time

Dispatch Protocol: Supported Dispatch Protocols

Execution Time: Time

-- Protocol support

Provided Virtual Bus Class : **inherit list of classifier** (virtual bus)

Provided Connection Quality Of Service : **inherit list of** Supported Connection QoS

~~-- mode related properties~~

~~Resumption Policy: **enumeration** (restart, resume)~~

~~Deactivation Policy: **enumeration** (inactive, unload) => inactive~~

~~-- need for and provision of address space protection~~

~~Runtime Protection : **inherit aadlboolean**~~

~~Runtime Protection Support : **aadlboolean**~~

~~-- Virtual machine layering~~
~~Implemented As: **classifier** (system implementation, abstract implementation)~~

Semantics

(34) A virtual processor is the logical execution platform component that is capable of scheduling and executing threads and other virtual processors. Threads and virtual processors will be bound to a virtual processor or processor for their execution. The `Allowed_Dispatch_Protocol` property specifies the dispatch protocols that a virtual processor supplies, i.e., only threads or virtual processors whose dispatch protocol is supported can be bound.

(35) Support for thread execution may require software that implements virtual processor functionality such as thread scheduling, e.g., an operating system kernel or other software virtual machine. Such software must be bound to a memory component that is accessible to the processor via the `Actual_Memory_Binding` property. Services provided by such software can be called through the provides subprogram access features of a virtual processor.

(36) A virtual processor component can include protocols in its abstraction. These protocols can be explicitly modeled as virtual bus subcomponents to satisfy protocol requirements by connections. The fact that a protocol is supported by a virtual processor can also be specified by a `Provided_Virtual_Bus_Class` property.

(37) A virtual processor can be declared as a subcomponent of a processor or another virtual processor; it can also be declared separately in a system component and then bound to a processor or another virtual processor. The difference between the two uses of virtual processor is that in the case of the subcomponent of a processor or virtual processor the binding of the virtual processor is implicit in the containment relationship.

(38) Flow specifications model logical flow through virtual processors. For example, they may represent requests for operating system services through subprograms or ports.

(39) Modes allow virtual processor components to have different property values under different operational virtual processor modes. Modes may be used to specify different runtime selectable configurations of virtual processor implementations.

3.3 Memory

(40) A memory component represents an execution platform component that stores code and data binaries. This execution platform component consists of hardware such as randomly accessible physical storage, e.g., RAM, ROM, or more complex permanent storage such as disks, reflective memory, or logical storage. Memories have properties such as the number and size of addressable storage locations. Subprograms, data, and processes – reflected in binary images - are bound to memory components for access by processors when executing threads. A memory component may be contained in a processor or may be accessible from a processor via a bus.

Legality Rules

Category	Type	Implementation
memory	Features • requires bus access • provides bus access • requires virtual bus access • provides virtual bus access • feature group • feature • port Flow specifications: no Modes: yes Properties: yes	Subcomponents: • memory • bus • abstract Subprogram calls: no Connections: yes Flows: no Modes: yes Properties: yes

(L27) A memory type can contain virtual bus access and bus access declarations, feature groups, a modes subclause, and property associations. It must not contain flow specifications.

(L28) A memory implementation can contain abstract, memory, and bus subcomponent declarations.

(L29) A memory implementation can contain a modes subclause and property associations.

(L30) A memory implementation can contain bus access connection declarations. Bus access connections can connect a memory subcomponent to a bus subcomponent or a requires bus access feature, as well as connect a provides bus access feature to a bus subcomponent.

(L31) A memory implementation must not contain flows subclause, or subprogram calls subclause.

~~Standard Properties~~

```
-- Properties related memory as a resource and its access
Memory_Protocol: enumeration (execute_only, read_only, write_only, read_write) => read_write
Word_Size: Size => 8 bits
Byte_Count: aadlinteger 0 .. Max_Byte_Count
Memory_Size: Size
Word_Space: aadlinteger 1 .. Max_Word_Space => 1
Base_Address: aadlinteger 0 .. Max Base_Address
Read_Time: record (
   Fixed: Time_Range;
   PerByte: Time_Range; )
Write_Time: record (
   Fixed: Time_Range;
   PerByte: Time_Range; )
-- Hardware description properties
Hardware_Description_Source_Text: inherit list of aadlstring
Hardware_Source_Language: Supported_Hardware_Source_Languages
-- mode related properties
Resumption_Policy: enumeration ( restart, resume )
-- Virtual machine layering
```

~~Implemented As: **classifier** (system implementation, abstract implementation)~~

Semantics

(41) Memory components are used to store binary images of source text, i.e., code and data. These images are loaded into memory representing the virtual address space of a process and are accessible to threads contained in the respective processes bound to the processor. Such access is possible if the memory is contained in this processor or is accessible to this processor via a shared bus component. Loading of binary images into memory may occur during processor startup or the binary images may have been preloaded into memory before system startup. An example of the latter case is PROM or EPROM containing binary images.

(42) A memory is accessible from a processor if the memory is contained in a processor or is connected via a shared bus component and the `Allowed_Physical_Access` property value for that bus includes `Memory_Access`.

(43) Memory components can have different property values under different operational modes.

(44) The memory component is intended to be an abstraction of a physical storage component. This can be a memory component such as RAM, or it can represent a more abstract storage component such as a map database. If it is desirable, the internals of the memory can be modeled by AADL as a system in its own right, i.e., an application system and an execution platform. For example, a map data base as a memory component can be modeled as a set of processors and disks as well as the database software. This system implementation description can be associated with the memory component declaration by the `Implemented_As` property (see Section 14).

(45) Ports can be used as mode transition triggers in memory with modes.

3.4 Buses

(46) A bus component represents an execution platform component that can exchange control and data between memories, processors, and devices. This execution platform component represents a communication channel, typically hardware together with communication protocols. Supported communication protocols can be explicitly modeled through virtual buses (see Section 6.5).

(47) Processors, devices, and memories can communicate by accessing a shared bus. Such a shared bus can be located in the same system implementation as the execution platform components sharing it or higher in the system hierarchy. Memory, processor, and device types, as well as the system type of systems they are contained in, can declare a need for access to a bus through a requires bus reference.

(48) Buses can be connected directly to other buses by one bus requiring access to another bus. Buses connected in such a way can have different bus classifier references.

(49) Connections between software components that are bound to different processors transmit their information across buses whose protocol supports the respective connection category.

Legality Rules

Category	Type	Implementation
bus	Features • requires bus access • provides bus access • requires virtual bus access • provides virtual bus access • feature group • feature • port Flow specifications: no Modes: yes Properties: yes	Subcomponents: • virtual bus • abstract Subprogram calls: no Connections: no Flows: no Modes: yes Properties: yes

(L32) A bus type can have can have virtual bus and bus access declarations, a modes subclause, and property associations.

(L33) A bus type must not contain any flow specifications.

(L34) A bus implementation can contain virtual bus and abstract subcomponent declarations.

(L35) A bus implementation can contain a modes subclause and property associations.

(L36) A bus implementation must not contain flows subclause, or subprogram calls subclause.

Standard Properties

```
-- Properties specifying bus transmission properties

Allowed Connection Type: list of enumeration
                         (Sampled Data Connection, Immediate Data Connection,
                          Delayed Data Connection, Port Connection,
                          Data Access Connection,
                          Subprogram Access Connection)
_
Allowed Physical Access Class: list of classifier ( device, processor, memory, bus )
Allowed Physical Access: list of reference ( device, processor, memory, bus )
Allowed Message Size: Size Range

Transmission Type: enumeration ( push, pull )

Transmission Time: record (

   Fixed: Time Range;

   PerByte: Time Range; )

Prototype Substitution Rule: inherit enumeration (Classifier_Match, Type_Extension,
Signature Match)

-- Hardware description properties

Hardware Description Source Text: inherit list of aadlstring

Hardware Source Language: Supported Hardware Source Languages


Access_Right : Access_Rights => read_write

-- Protocol support

Provided Virtual Bus Class : inherit list of classifier (virtual bus)

Provided Connection Quality Of Service : inherit list of Supported Connection QoS

-- mode related properties

Resumption Policy: enumeration ( restart, resume )

-- Virtual machine layering

Implemented As: classifier ( system implementation, abstract implementation )
```

Semantics

(50) A bus support physical communication between processors, memories, and devices. This allows a processor to support execution of source text in the form of code and data loaded as binary images into memory components. A bus allows a processor to access device hardware when executing device software. A bus may also support different port and subprogram connections between thread components bound to different processors. The `Allowed_Connection_Type` property indicates which forms of access a particular bus supports. The bus may constrain the size of messages communicated through data or event data connections.

(51) A bus component provides access between processors, memories, and devices. It is a shared component, for which access is required by each of the respective components. A device is accessible from a processor if the two share a bus component. A memory is accessible from a processor if the two share a bus component.

(52) Buses can be directly connected to other buses. This is represented by one bus declaration specifying access to another bus in its requires subclause.

(53) Physical access to a bus can be restricted to certain types of devices, memory, buses, and processors. This is achieved with the property `Allowed_Physical_Access`.

(54) Bus components can have different property values under different operational modes.

(55) A bus component can include protocols in its abstraction. Protocols provided by a bus can be specified with the `Provided_Virtual_Bus_Class` property. They are matched against protocol requirements of connections and virtual buses as specified by their `Required_Virtual_Bus_Class` property. If desired instances of protocols supported by a bus can be explicitly modeled as virtual bus subcomponents. In that case the connection or virtual bus requiring a specific protocol can be bound to the specific virtual bus instance.

(56) Virtual buses (protocols) may be implemented in the bus hardware or in software. The virtual bus software executes on processors connected to the bus, whose bound threads communicate over connections that require the protocol.

(57) The bus is intended to be an abstraction of a physical bus or network. If it is desirable, the internals of the bus can be modeled by AADL as a system in its own right, i.e., an application system and an execution platform. This system implementation description can be associated with the bus component declaration by the `Implemented_As` property (see Section 14).

(58) Ports can be used as mode transition triggers in buses with modes.

Processing Requirements and Permissions

(59) A method of implementation shall define how the final size of a transmission is determined for a specific connection. Implementation choices and restrictions such as packetization and header and trailer information are not defined in this standard.

(60) A method of implementation shall define the methods used for bus arbitration and scheduling. If desired this can be modeled by a notation of your choice and associated with a bus via property. Alternatively, it can be modeled through an AADL system model and associated with the bus through an `Implemented_As` property.

Examples

3.5 Virtual Buses

(61) A virtual bus component represents logical bus abstraction such as a virtual channel or communication protocol.

(62) Virtual buses can be bound to buses, virtual buses, processors, virtual processors, devices, or memory. When bound to a bus, it may represent multiple protocols supported by the bus or a virtual channel with a subset of the bus bandwidth. When bound to a virtual bus it may represent a hierarchy of protocols or virtual channels. When bound to a processor it may represent protocols supported by a processor. When bound to a sequence of hardware components it may represent a virtual channel or flow across these components.

(63) Virtual buses can be connected to each other, to virtual processors, and to devices. This allows users to model virtual platforms as a collection of virtual components.

(64) Virtual buses can be subcomponents of processors, buses, and other virtual buses. This implies that they are bound to the processor, bus, or virtual bus they are contained in.

(65) Connections and virtual buses can indicate by property the protocols they require by identifying the appropriate virtual bus or bus classifier. A connection can also indicate the desired level of quality of service, which must be satisfied

by the virtual bus or bus the connection is bound to. Similarly, hardware components can indicate by property the virtual buses they provide.

Legality Rules

Category	Type	Implementation
virtual bus	Features • port • provides virtual bus access • requires virtual bus access • feature • feature group Flow specifications: no Modes: yes Properties: yes	Subcomponents: • virtual bus • abstract Subprogram calls: no Connections: no Flows: no Modes: yes Properties: yes

(L37) A virtual bus type can have property associations.

(L38) A virtual bus type must not contain flow specifications.

(L39) A virtual bus implementation can contain virtual bus subcomponent declarations.

(L40) A virtual bus implementation can contain a modes subclause and property associations.

(L41) A virtual bus implementation must not contain a connections subclause, flows subclause, or subprogram calls subclause.

Consistency Rules

(C4) In a fully deployed system virtual buses must be directly or indirectly bound to processors or buses that support these virtual buses, or they must be subcomponents of buses and processors.

(C5) There must not be cycles in binding declarations between virtual buses.

~~*Standard Properties*~~

~~-- Properties specifying bus transmission properties~~

~~Allowed Connection_Type: **list of enumeration** (Sampled_Data_Connection, Immediate_Data_Connection, Delayed_Data_Connection, Port_Connection, Data_Access_Connection, Subprogram_Access_Connection)~~

~~Allowed Message_Size: Size_Range~~

~~Transmission_Type: **enumeration** (push, pull)~~

~~Transmission_Time: **record** (~~

~~Fixed: Time_Range;~~

~~PerByte: Time_Range;)~~

~~Prototype_Substitution_Rule: **inherit enumeration** (Classifier_Match, Type_Extension, Signature_Match)~~

~~-- Protocol_support~~

~~Provided_Connection_Quality_Of_Service : **inherit list of** Supported_Connection_QoS~~

~~Provided_Virtual_Bus_Class : **inherit list of classifier** (virtual_bus)~~

~~Required Connection Quality Of Service : **inherit list of** Supported Connection QoS~~

~~Allowed Connection Binding Class:~~
~~**inherit list of classifier**(processor, virtual processor, bus, virtual bus, device, memory, system)~~

~~Allowed Connection Binding: **inherit list of reference** (processor, virtual processor, bus, virtual bus, device, memory, system)~~

~~Actual Connection Binding: **inherit list of reference** (processor, virtual processor, bus, virtual bus, device, system, memory)~~

~~Required Virtual Bus Class : **inherit list of classifier** (virtual bus)~~

~~-- mode related properties~~

~~Resumption Policy: **enumeration** (restart, resume)~~

~~-- Virtual machine layering~~

~~Implemented As: **classifier** (system implementation, abstract implementation)~~

Semantics

(66) A virtual bus represents a virtual channel or communication protocol. The `Provided_Virtual_Bus_Class` property is used to indicate that a processor or bus supports a protocol. Similarly, the `Required_Virtual_Bus_Class` property is used to indicate that a protocol is required by a connection or virtual bus.

(67) A virtual bus can be a subcomponent of a virtual bus, bus, virtual processor, processor, or system, it can be provided as indicated by the `Provided_Virtual_Bus_Class` property of a bus, virtual bus, virtual processor, or processor. In all cases, this indicates that the protocol represented by the virtual bus is supported on the bus or processor.

(68) If a virtual bus requires another virtual bus as expressed by the `Required_Virtual_Bus_Class` property, this required access is satisfied by binding the protocol to a processor or bus that provides this virtual bus as specified with the `Provided_Virtual_Bus_Class` property.

(69) A virtual bus can represent a portion of the bandwidth capacity of a bus. It can act as virtual channel that can make certain performance guarantees.

(70) The `Allowed_Connection_Type` property indicates which forms of access a particular virtual bus supports. The virtual bus may constrain the size of messages communicated through data or event data connections.

(71) Virtual bus components can have different property values under different operational modes.

(72) If it is desirable, the internals of the virtual bus can be modeled by AADL as an application system in its own right, e.g., as a sender thread interacting with a receiver thread. This system implementation description can be associated with the virtual bus component declaration by the `Implemented_As` property (see Section 14).

Processing Requirements and Permissions

(73) Ports can be used as mode transition triggers in virtual buses with modes.

(74) A method of implementation shall define how the final size of a transmission is determined for a specific connection. Implementation choices and restrictions such as packetization and header and trailer information are not defined in this standard.

(75) A method of implementation shall define the methods used for bus arbitration and scheduling. If desired this can be modeled by a notation of your choice and associated with a virtual bus via a property. Alternatively, it can be modeled through an AADL system model and associated with the bus through an `Implemented_As` property.

Examples

3.6 Devices

(76) A device component represents a physical or mechanical component within the system, entities in the external environment. A device is logically connected to application software components and physically connected to computing hardware. Examples of devices are sensors, actuators, cameras, brakes. A device may represent a physical entity or its (simulated) software equivalent. A device may exhibit simple or complex behaviors. If the device has associated software such as device drivers that must reside in a memory and execute on a processor external to the device, this can be specified through appropriate property values for the device.

(77) A device interacts with both execution platform components and application software components. A device has logical connections to application software components. Those logical connections are represented by connection declarations between device ports and application software component ports. A device also has physical connections to processors via a bus. This models software executing on a processor accessing the physical device. For any logical connection between a device and a thread executing application source code, there must be a physical connection in the execution platform.

(78) A device can be viewed to be as part of the application system. In this case, it is natural to place the device together with the application software components. The physical connection to processors must follow the system hierarchy.

(79) Alternatively a device may be viewed to be part of the execution platform. In this case, it is placed in proximity of other execution platform components. The logical connections have to follow the system hierarchy to connect to application software components.

(80) Examples of devices are sensors and actuators that interface with the external physical world, or standalone physical systems (such as a GPS) or dedicated hardware (such as counters or timers). Devices communicate with embedded application systems through ports and connections and with the computing hardware through bus access.

(81) Devices can be part of virtual platforms, i.e., they can be connected to other virtual platform components, such as virtual, and virtual buses through virtual bus access connections.

Legality Rules

Category	Type	Implementation
device	Features • port • feature group • provides subprogram access • provides subprogram group access • requires bus access • provides bus access • provides virtual bus access • requires virtual bus access • feature Flow specifications: yes Modes: yes Properties: yes	Subcomponents: • bus • virtual bus • data • abstract Subprogram calls: no Connections: yes Flows: yes Modes: yes Properties: yes

(L42) A device type can contain port, feature group, provides subprogram access, provides subprogram group access, bus access declarations, flow specifications, a modes subclause, as well as property associations.

(L43) A device component implementation must not contain a subprogram calls subclause.

(L44) A device implementation can contain abstract, data, virtual bus, and bus subcomponents, bus access connections, a modes subclause, a flows subclause, and property associations.

Standard Properties

-- Hardware description properties

Hardware Description Source Text: **inherit list of aadlstring**

Hardware Source Language: Supported Hardware Source Languages

-- Properties specifying device driver software that must be

-- executed by a processor

Source Text: **inherit list of aadlstring**

Source Language: **inherit list of** Supported Source Languages

Code Size: Size

Data Size: Size

Stack Size: Size

-- Properties specifying the execution properties of the device or its driver

Dispatch Protocol: Supported Dispatch Protocols

Dispatch Trigger: **list of reference** (port)

Period: **inherit** Time

Compute Execution Time: Time Range

Deadline: **inherit** Time => Period

Reference Time: **inherit reference** (processor, device, bus, system, abstract)

-- scheduling properties

Time Slot: **list of aadlinteger**

Priority: **inherit aadlinteger**

-- Properties specifying constraints for processor and memory binding

-- for the device driver software

Allowed Memory Binding Class:

inherit list of classifier (memory, system, processor, virtual processor)

Allowed Memory Binding: **inherit list of reference** (memory, system, processor, virtual processor)

Actual Memory Binding: **inherit list of reference** (memory, system, processor, virtual processor)

Actual Processor Binding: **inherit list of reference** (processor, virtual processor, device, system)

Allowed Processor Binding Class:

inherit list of classifier (processor, virtual processor, device, system)

Allowed Processor Binding: **inherit list of reference** (processor, virtual processor, device, system)

-- protocol support

Provided Virtual Bus Class : **inherit list of classifier** (virtual bus)

Provided Connection Quality Of Service : **inherit list of** Supported Connection QoS

-- mode related properties

Resumption Policy: **enumeration** (restart, resume)

~~-- Virtual machine layering~~

~~Implemented As: **classifier** (system implementation, abstract implementation)~~

Semantics

(82) AADL device components model dedicated hardware or physical entities in the external environment, e.g., a GPS system, or entities that interface with an external environment, e.g., sensors and actuators as interface between a physical plant and a control system. Devices may represent a physical entity or its (simulated) software equivalent. They may exhibit simple behavior, e.g., a timer, sensor, or actuator, or complex behaviors, e.g., a camera, GPS, or an engine. In the latter case sensors or actuators may also be represented by device ports. Devices are logically connected to application software components and physically connected to processors. The device functionality may be fully embedded in the device hardware, or it may be provided by device-specific driver software.

(83) A device is accessible from a processor if the device is connected via a shared bus component.

(84) A device declaration can include flow specifications that indicate that a device is a flow source, a flow sink, or a flow path exists through a device.

(85) Device components can have different property values under different operational modes.

(86) A device component can contain a data subcomponent to represent persistent state. This data subcomponent cannot be made accessible via data access. Device behavior can be specified via Behavior Annex subclauses, which can refer to the data subcomponent.

(87) Embedded application software interacts with devices through port connections and through subprogram calls. Data ports can be used to represent device registers. Event and event data ports can be used to represent signals and interrupts that trigger execution of software or that initiate a reaction by the device. Subprogram calls reflect functionality executed by the device. This functionality may be fully implemented in the device hardware or through a device driver.

(88) The `Dispatch_Protocol` property determines the execution behavior of the device. A device can execute periodically or event-driven. Periodic execution means that the device polls the external environment periodically to produce a periodic data stream to the application, or that it samples input from the application periodically. Event-driven execution means that the device generates events, e.g., a clock or timer, that it observes events in the external environment and passes them to the application, e.g., flipping of a switch, or that it responds to events or event data being sent by the application, e.g., a signal to turn on a light. The input or output rate on device ports can be specified through the `Output_Rate` property. The `Dispatch_Trigger` property can be used to specify a subset of event or event data ports that can trigger a device dispatch.

(89) The interface to a device may be through a device driver. The features of the device type may represent the interface to the device via the device driver. The execution behavior of the device driver is specified by the device dispatch protocol.

(90) The device driver may execute as part of the underlying operating system kernel. In this case, we can specify the driver characteristics as properties on the device itself, such as the code size, data size, and stack size. Binding properties specify the processor whose runtime system includes the driver.

(91) The device driver may execute in a separate address space from the operating system kernel. In this case, the binding property may specify a virtual processor as target.

(92) A device driver may execute as an application thread. In this case, the driver is modeled by an AADL thread. This thread provides the device interface to the application and interfaces with the device registers of the physical device, represented by the AADL device component.

(93) A device can contain a bus subcomponent that it makes externally accessible. This models a bus that is managed by the device and other components can connect to it. In this case the device is implicitly connected to this bus.

(94) A device can support communication protocols as expressed by the `Provided_Virtual_Bus_Class` property. Instances of such protocols can also be explicitly represented by virtual bus subcomponents.

(95) The device is intended to be an abstraction of a physical component in the embedded system environment. If it is desirable, the internals of the device can be modeled by AADL as a system in its own right, i.e., an application system and an execution platform. For example, a digital camera as a device can be modeled as a set of processors and application software that handles the taking of images and their transfer from the camera to a processor via a USB bus connection. This system implementation description can be associated with the device component declaration by the `Implemented_As` property (see Section 14).

Processing Requirements and Permissions

(96) This software must reside as a binary image on memory components and is executed on a processor component. The executing processor that has access to the device must be connected to the device via a bus. The memory storing the binary image must be accessible to the processor. Device driver software may be modeled implicitly through the `Source_Text` and related properties, or it may be modeled explicitly by a separate thread declaration.

Examples

4. SYSTEM COMPOSITION

(97) Systems are organized into a hierarchy of components to reflect the structure of actual systems being modeled. This hierarchy is modeled by *system* declarations to represent a composition of components into composite components. A *system instance* models an instance of an application system and its binding to a system that contains execution platform components.

4.1 Systems

(98) A system represents an assembly of interacting application software, execution platform, and system components. Systems can have multiple modes, each representing a possibly different configuration of components and their connectivity contained in the system. Systems may require access to data and bus components declared outside the system and may provide access to data and bus components declared within. Systems may be hierarchically nested.

Legality Rules

Category	Type	Implementation
system	Features: • port • feature group • provides subprogram access • requires subprogram access • provides subprogram group access • requires subprogram group access • provides bus access • requires bus access • provides virtual bus access • requires virtual bus access • provides data access • requires data access • feature Flow specifications: yes Modes: yes Properties: yes	Subcomponents: • data • subprogram • subprogram group • process • processor • virtual processor • memory • bus • virtual bus • device • system • abstract Subprogram calls: no Connections: yes Flows: yes Modes: yes Properties: yes

(L45) A system component type can contain subprogram, subprogram group, data and bus access declarations, port, feature group declarations. It can also contain flow specifications as well as property associations.

(L46) A system component implementation can contain abstract, data, subprogram, subprogram group, process, and system subcomponent declarations as well as execution platform components, i.e., processor, virtual processor, memory, bus, virtual bus, and device.

(L47) A system implementation can contain a modes subclause, a connections subclause, a flows subclause, and property associations.

(L48) A thread group must not contain a subprogram calls subclause.

Standard Properties

-- Properties related to source text

Source_Text: **inherit list of aadlstring**

Source_Language: **inherit list of** Supported_Source_Languages

-- Process property that can be specified at the system level as well

-- Runtime enforcement of address space boundaries

Runtime_Protection : **inherit aadlboolean**

-- Inheritable thread properties

Synchronized_Component: **inherit aadlboolean => true**

Active_Thread_Handling_Protocol:
  inherit Supported_Active_Thread_Handling_Protocols => abort
Period: **inherit** Time

Deadline: **inherit** Time => Period

-- execution time related properties

Reference_Processor: **inherit classifier** (processor)

-- scheduling properties

Time_Slot: **list of aadlinteger**

Priority: **inherit aadlinteger**

-- Properties related binding of software component source text in

-- systems to processors and memory

Allowed_Processor_Binding_Class:
  inherit list of classifier (processor, virtual processor, device, system)

Allowed_Processor_Binding: **inherit list of reference** (processor, virtual processor, device, system)

Actual_Processor_Binding: **inherit list of reference** (processor, virtual processor, device, system)

Allowed_Memory_Binding_Class:
  inherit list of classifier (memory, system, processor, virtual processor)

Allowed_Memory_Binding: **inherit list of reference** (memory, system, processor, virtual processor)

Actual_Memory_Binding: **inherit list of reference** (memory, system, processor, virtual processor)

Allowed_Connection_Binding_Class:
  inherit list of classifier(processor, virtual processor, bus, virtual bus, device, memory, system)

Allowed_Connection_Binding: **inherit list of reference** (processor, virtual processor, bus, virtual bus, device, memory, system)

~~Actual_Connection_Binding: **inherit list of reference** (processor, virtual processor, bus, virtual bus, device, system, memory)~~

~~Collocated: **record** (~~

~~Targets: **list of reference** (data, thread, process, system, connection);~~

~~Location: **classifier** (processor, memory, bus, system);)~~

~~Not_Collocated: **record** (~~

~~Targets: **list of reference** (data, thread, process, system, connection);~~

~~Location: **classifier** (processor, memory, bus, system);)~~

~~-- Properties related systems as execution platforms~~

~~Hardware_Source_Language: Supported_Hardware_Source_Languages~~

~~-- mode related properties~~

~~Resumption_Policy: **enumeration** (restart, resume)~~

~~-- Properties related to startup of processor contained in a system~~

~~Startup_Deadline: Time~~

~~Startup_Execution_Time: Time_Range~~

~~-- Properties related to system load times~~

~~Load_Time: Time_Range~~

~~Load_Deadline: Time~~

~~-- Properties related to the hardware clock~~

~~Clock_Jitter: Time~~

~~Clock_Period: Time~~

~~Scheduling_Protocol: **inherit list of** Supported_Scheduling_Protocols~~

~~Clock_Period_Range: Time_Range~~

Semantics

(99) A system component represents an assembly of software and execution platform components. All subcomponents of a system are considered to be contained in that system.

(100) Some system components consist of purely software components all of which must be bound to execution platform components outside the system itself. An example is an application software system. Some system components consist purely of computing hardware components. They represent aggregations of processor, memory, and bus components that act as the hardware platform. Some system component is a composition of devices and buses that represent the physical environment that the embedded software system interacts with. Some system components may be combinations of the above. Some system components are self-contained in that all contained software components are bound to execution platform components contained within the same system. Such self-contained systems may have external connectivity in the form of logical connection points represented by ports and physical connection points in the form of required or provided bus access. Examples, of such systems are database servers, GPS receivers, and digital cameras. Such self-contained systems with an external interface may represent the implementation of devices. The device representation takes a black-box perspective, while the system representation takes a white-box perspective and is associated with the device through the `Implemented_As` property.

(101) A system component can contain a modes subclause. Each mode can represent an alternative system configuration of contained subcomponents and their connections. The transition between modes is determined by the mode transition declarations of specific property associations.

Processing Requirements and Permissions

(102) Processing methods may restrict data, subprogram, and subprogram group subcomponents to be part of only one process address space. In that case they may require those subcomponents to be placed inside a process, thread group, or thread, and not be allowed in system implementations.