



AEROSPACE STANDARD

AS5506™/2

Issued 2011-01
Reaffirmed 2025-10

SAE Architecture Analysis and Design Language (AADL) Annex Volume 2:
Annex B: Data Model Annex
Annex D: Behavior Model Annex
Annex F: ARINC653

RATIONALE

The purpose of the annexes in this document are to provide guidance on how to associate data models expressed in other notations with an architecture model expressed in AADL, to enable modeling of component and component interaction behavior in a state-machine based annex sublanguage, and to provide guidance on how to model ARINC653 compliant partitioned embedded system architectures in AADL models.

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

AS5506/2 has been reaffirmed to comply with the SAE Five-Year Review policy.

SAE Executive Standards Committee Rules provide that: "This report is published by SAE to advance the state of technical and engineering sciences. The use of this report is entirely voluntary, and its applicability and suitability for any particular use, including any patent infringement arising therefrom, is the sole responsibility of the user."

SAE reviews each technical report at least every five years at which time it may be revised, reaffirmed, stabilized, or cancelled. SAE invites your written comments and suggestions.

Copyright © 2025 SAE International

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, or used for text and data mining, AI training, or similar technologies, without the prior written permission of SAE.

TO PLACE A DOCUMENT ORDER: Tel: 1-877-606-7323 (U.S. and Canada only)
Tel: 1-724-776-4970 (outside U.S. and Canada)
Fax: 724-776-0790
Email: CustomerService@sae.org
<http://www.sae.org>

SAE WEB ADDRESS:

Full document details: [SAE AS5506_2](#)

For more information on this standard, visit
<https://www.sae.org/standards/content/AS5506/2/>

Documents

Standard Sharing Hub

TABLE OF CONTENTS

1. INFORMATION AND FEEDBACK.....	6
2. REFERENCES	7
2.1 Normative References	7
2.2 Informative References	7
ANNEX DOCUMENT B DATA MODELING.....	8
Annex B.1 Scope.....	8
Annex B.2 Modeling Data Types in AADL	8
Annex B.3 Data Modeling Property Set.....	9
Annex B.4 Predeclared AADL Package for Basic Data Types	11
Annex B.5 Examples	14
ANNEX DOCUMENT D BEHAVIOR MODEL.....	17
Annex D.1 Scope.....	17
Annex D.2 Overview of Behavior Annex Concepts	17
Annex D.3 Behavior Specification	19
Annex D.4 Thread Dispatch Behavior Specification.....	24
Annex D.5 Component Interaction Behavior Specification	27
Annex D.6 Behavior Action Language.....	30
Annex D.7 Behavior Expression Language.....	34
Annex D.8 Synchronization Protocols	38
ANNEX DOCUMENT F ARINC653	42
Annex F.1 Scope.....	42
Annex F.2 ARINC653 Partition Management (ARINC653 Module).....	42
Annex F.3 ARINC653 Partitions Modeling.....	43
Annex F.4 Multi Processor Architectures Modeling	45
Annex F.5 ARINC653 Process Modeling.....	46
Annex F.6 ARINC653 Inter-partition Communication Modeling	47
Annex F.7 ARINC653 Intra-partition Communication Modeling	48
Annex F.8 Modeling of ARINC653 Buffer	49
Annex F.9 Modeling of ARINC653 Blackboards	49
Annex F.10 Modeling of ARINC653 Events	50
Annex F.11 Modeling of ARINC653 Semaphores.....	51
Annex F.12 ARINC653 Memory Requirements Modeling.....	52
Annex F.13 ARINC Health Monitor (HM) Modeling.....	52
Annex F.14 Modeling of ARINC653 Modes/States	53
Annex F.15 Modeling of Device Drivers	53
Annex F.16 Summary of Modeling Rules.....	55
Annex F.17 ARINC653 Property Set.....	57
Annex F.18 System Validation using the ARINC653 Annex	61
Annex F.19 Example with One Module	61
Annex F.20 Example with Two Modules	69

Table of Figures

Figure 1: Sender Behavior Specification.....	26
Figure 2: Client Server Behavior.....	40
Figure 3: ARINC653 Module (without partition runtime) as AADL Processor	43
Figure 4: Partition and its Association to the Runtime System as AADL Process and Virtual Processor.....	45
Figure 5: Multi-processor system model.....	45
Figure 6: ARINC653 Process inside a Partition as AADL Thread inside an AADL Process.....	47
Figure 7: ARINC653 Inter-partition Queuing Port Connection as AADL Event Data Port Connection	48
Figure 8: ARINC653 Inter-partition Sampling Port Connection as AADL Data Port Connection	48
Figure 9: ARINC653 Buffer Communication Channel as AADL Event Data Port Connection	49
Figure 10: ARINC653 Blackboard Communication Channel as Data Port Connection (or Shared Data)	50
Figure 11: ARINC653 Event Communication Channel as AADL Event Port Connection	51
Figure 12: ARINC653 Semaphore Usage as AADL Shared Data Component Access	51
Figure 13: Binding of ARINC653 Partitions to Memory Segments as AADL Process Binding to Memory	52
Figure 14: Modeling of Devices & Drivers in ARINC653 Architectures	54
Figure 15: Intra- and Inter-partition Communication Example.....	62
Figure 16: Two Module Example	69

FOREWORD

- (1) The AADL standard was prepared by the SAE Avionics Systems Division (ASD) Embedded Computing Systems Committee (AS-2) Architecture Description Language (AS-2C) subcommittee.
- (2) This AADL standard defines three annexes to the SAE AADL standard AS5506A.
- (3) This AADL standard includes a Data Modeling Annex that defines a property set, a package with pre-declared basic data types, and guidance on how to map a data model expressed in a data modeling language into an architecture model expressed in AADL.
- (4) This AADL standard includes a Behavior Annex that extends the AADL core language with a state machine-based sublanguage annex notation for specifying component and component interaction behavior.
- (5) This AADL standard includes an ARINC653 Annex that that defines a property set and guidance on how to model ARINC653 compliant partitioned embedded system architectures in AADL.

INTRODUCTION

- (1) The SAE Architecture Analysis & Design Language (referred to in this document as AADL) is a textual and graphical language used to design and analyze the software and hardware architecture of performance-critical real-time systems. These are systems whose operation strongly depends on meeting non-functional system requirements such as reliability, availability, timing, responsiveness, throughput, safety, and security. AADL is used to describe the structure of such systems as an assembly of software components mapped onto an execution platform. It can be used to describe functional interfaces to components (such as data inputs and outputs) and performance-critical aspects of components (such as timing). AADL can also be used to describe how components interact, such as how data inputs and outputs are connected or how application software components are allocated to execution platform components. The language can also be used to describe the dynamic behavior of the runtime architecture by providing support to model operational modes and mode transitions. The language is designed to be extensible to accommodate analyses of the runtime architectures that the core language does not completely support. Extensions can take the form of new properties and analysis specific notations that can be associated with components and are standardized themselves.
- (2) AADL was developed to meet the special needs of performance-critical real-time systems, including embedded real-time systems such as avionics, automotive electronics, or robotics systems. The language can describe important performance-critical aspects such as timing requirements, fault and error behaviors, time and space partitioning, and safety and certification properties. Such a description allows a system designer to perform analyses of the composed components and systems such as system schedulability, sizing analysis, and safety analysis. From these analyses, the designer can evaluate architectural tradeoffs and changes.
- (3) AADL supports analysis of cross cutting impact of change in the architecture along multiple analysis dimensions in a consistent manner. Consistency is achieved through automatic generation of analysis models from the annotated architecture model. AADL is designed to be used with generation tools that support the automatic generation of the source code needed to integrate the system components and build a system executive from validated models. This architecture-centric approach to model-based engineering permits incremental validation and verification of system models against requirements and implementations against systems models throughout the development lifecycle.
- (4) This document consists of three annexes to the SAE AADL standard that
 - provides guidance on a standard way of associating data models expressed in other data modeling notations such as UML or ASN.1 with architecture models expressed in AADL,
 - enables modeling of component and component interaction behavior in a state-machine based annex sublanguage, and
 - provides guidance on a standard way of representing ARINC653 standard compliant partitioned embedded system architectures in AADL models.

1. INFORMATION AND FEEDBACK

- (1) The website at <http://www.aadl.info> is an information source regarding the SAE AADL standard. The website provides information and a download site for the Open Source AADL Tool Environment. It also provides links to other resources regarding the AADL standard and its use.
- (2) The public AAD Wiki (<https://wiki.sei.cmu.edu/aadl>) maintains a list of AADL related publications by the community, and provides guidance on the use of extension of the open source OSATE tool set for AADL.
- (3) Questions and inquiries regarding working versions of annexes and future versions of the standard can be addressed to info@aadl.info.
- (4) Informal comments on this standard may be sent via e-mail to errata@aadl.info. If appropriate, the defect correction procedure will be initiated. Comments should use the following format:

!topic Title summarizing comment

!reference AADL-ss.ss(pp)

!from Author Name yy-mm-dd

!keywords keywords related to topic

!discussion

text of discussion

- (5) where ss.ss is the section, clause or subclause number, pp is the paragraph or line number where applicable, and yy-mm-dd is the date the comment was sent. The date is optional, as is the !keywords line.
- (6) Multiple comments per e-mail message are acceptable. Please use a descriptive "Subject" in your e-mail message.
- (7) When correcting typographical errors or making minor wording suggestions, please put the correction directly as the topic of the comment; use square brackets [] to indicate text to be omitted and curly braces { } to indicate text to be added, and provide enough context to make the nature of the suggestion self-evident or put additional information in the body of the comment, for example:

!topic [c]{C}haracter

!topic it["]s meaning is not defined

2. REFERENCES

The following publications form a part of this document to the extent specified herein. The latest issue of SAE publications shall apply. The applicable issue of the other publications shall be the issue in effect on the date of the purchase order. In the event of conflict between the text of this document and references cited herein, the text of this document takes precedence. Nothing in this document, however, supersedes applicable laws and regulations unless a specific exemption has been obtained.

2.1 Normative References

The following normative documents contain provisions that, through reference in this text, constitute provisions of this standard.

2.1.1 SAE Publications

Available from SAE International, 400 Commonwealth Drive, Warrendale, PA 15096-0001, Tel: 877-606-7323 (inside USA and Canada) or 724-776-4970 (outside USA), www.sae.org.

AS5506A Architecture Analysis & Design Language (AADL)

AS5506/1 Architecture Analysis & Design Language (AADL) Annex Volume 1

2.1.2 IEEE Publications

Available from Institute of Electrical and Electronics Engineers, 445 Hoes Lane, Piscataway, NJ 08854-4141, Tel: 732-981-0060, www.ieee.org.

IEEE/ANSI 610.12-1990 IEEE Standard Glossary of Software Engineering Terminology

2.1.3 Other Normative References

Unified Modeling Language Specification [UML 2007, version 2.1.1], August 2007, version 2.1.1.

ARINC653 Standard, AERONAUTICAL RADIO, INC., 2005.

2.2 Informative References

The following informative references contain background information about the items with the citation.

[BNF 1960] NAUR, Peter (ed.), "Revised Report on the Algorithmic Language ALGOL 60," *Communications of the ACM*, Vol. 3 No. 5, pp. 299-314, May 1960.

Annex Document B Data Modeling

Normative

Annex B.1 Scope

- (1) The Data Modeling Annex document provides guidelines to model data types as part of an AADL architectural model and guidelines to map data models expressed in other modeling notations into an architecture model expressed in AADL.
- (2) The purpose of AADL is to model the computer based embedded system, including the runtime architecture of embedded systems. In that context AADL provides the component category data, whose role is to represent data components with shared access, the data type of such data components, as well as the data type of data communicated through data ports, event data ports, and subprogram parameters. AADL supports implementation declarations for data components, which can represent the elements of a data type as subcomponents. Its intent is to represent the composition of a data component out of elements when relevant to the architecture, such as the aggregation of output from several sources to be communicated as a single aggregate, or concurrent access to shared data managed at the level of elements of a data component
- (3) AADL allows one to capture some abstractions of the data model. The core language standard specifies that by default the data type name in the source text is assumed to be the same as the identifier of the AADL data component. If necessary, the `Source_Name` property can be used to define a mapping from the AADL identifier to a name in the source text. Other information about the data types and data components in the source text can be captured through other properties, such as `Source_Data_Size` to indicate the amount of memory an instance of the data type takes. It is expected that there is an equivalent data type in the source text expressed in a source language such as C or Ada, in an equivalent notation for modeling frameworks like SCADE, SDL or Simulink, or data modeling notations (e.g. OMG IDL or ITU-T ASN.1).
- (4) In some occasions, users may want to record data modeling related information that is relevant for the runtime architecture as part of an AADL model to support checking of architectural consistency, consistency between architecture information and the source text, and generation of a runtime system from AADL models. The Data Modeling annex document is intended to support these objectives together with the Code Generation annex document.

Annex B.2 Modeling Data Types in AADL

- (1) The Data Model annex document provides guidelines for data modeling in the following way. User-defined AADL data component types represent application data types. These data component types are then annotated with properties to indicate relevant details of the data type in a data modeling language or source language. The property `Data_Representation` specifies the realization of the data type in terms of basic data types built into the language, such as `real`, `integer`, `string`, and `Boolean`, and in terms of composite data types, such as `array` (ordered sets accessible by index), `struct` (collection of named elements), and `union` (collection of named alternatives).
- (2) In the case of `array` or `string`, additional properties characterize the size of the array in terms of multiple dimensions (`Dimension`) and the data type of the array elements (`Base_Type`). Note that the type of the array elements is characterized in terms of an AADL defined data component type. This allows arrays of user-defined types as well as base types to be modeled. The Data Modeling Annex includes a package (`Base_Types`) of predeclared AADL data component types (and implementations) for a set of data types that represent built-in data types. Those data component types are themselves characterized by the `Data_Representation` property as representing a built-in data types such as `integer`, `fixed`, `float`, `string`, `character`, `enum`, `Boolean`.
- (3) In the case of `struct` or `union`, the designer has two options. First option is to specify the data types of the elements by an ordered list of classifiers as the `Base_Type` property value. The order of the list determines the order of the data elements. If the names of the data elements are relevant, then they can be specified by the `Element_Names` property as an ordered list of names. In this case, the designer only specifies which data are available, there is no mechanism to reference or interact with entities inside this data component at model level.

- (4) Second option is to use AADL mechanism to provide access to data type members or access methods to the data type represented by the data model. Data type members are expressed as subcomponents and access methods are expressed as provides subprogram access features in the AADL data type and implementation declaration.. Explicitly declaring members as data subcomponents gives more control on the architectural mapping of the members, such as memory layout, or access policies.

Note: these two options are notionally equivalent. No matter which option is chosen in the declarative AADL model, the second option (use of subcomponents) is the one used internally in the AADL processing tools to represent and manipulate arrays and structures. This allows treating both options in a uniform way.

- (5) The following properties further characterize representation of numeric values: `Data_Digits`, `Data_Scale` for specifying the precision of fixed-point types, `IEEE754_Precision` for float types and `Number_Representation` for integer types. Three other properties are intended to specify constraints on the numerical data, namely range constraints (`Real_Range`, `Integer_Range`) and the measurement unit in which the data values are intended to be interpreted (`Measurement_Unit`).
- (6) The `Code_Set` property allows for the specification of the character set to be used in string and character data types.
- (7) The `Enumerators` property allows for the specification of the literal values to be used in an enumeration data type.
- (8) The low-level representation of Enumerators can be defined using the `Representation` property.
- (9) The `Initial_Value` property allows for specification of the initial value of a data component.

Note: AADLV2 proposes standard properties to define data types. For instance, the fact that a data component in the source text represents a constant is recorded by the `Access_Right` property with the value `Read_Only`. Other properties exist to specify the memory requirements of a data type, the concurrency protocol to access it, etc.

- (10) An implementation method may restrict the combinations of properties from this property set, and the property from the AADL core.

Note: the rationale is to ensure there is one single definition of a data type, either as defined using this annex, or using core mechanisms with `Source_Language`, `Type_Source_Name`.

Annex B.3 Data Modeling Property Set

property set `Data_Model` is

`Base_Type` : **list of classifier** (`data`)

applies to (`data`);

The `Base_Type` property specifies the base type of a data component type. The classifiers being referenced are those defined in the `Base_Types` package or from user defined packages.

`Code_Set` : **aadlinteger**

applies to (`data`);

The `Code_Set` property is used to specify the code set used to represent a character or a string. The value applied is the registered value affected and defined in the “OSF character and Code Set Registry” by the OSF. This document is available at <http://www.opengroup.org/dce/info/>.

`Data_Digits` : **aadlinteger**

applies to (`data`);

The `Data_Digits` property specifies the total number of digits of a fixed-point type.

`Data_Scale` : **aadlinteger**

applies to (`data`);

The <code>Data_Scale</code> property defines the scale of the fixed-point types ($10^{**}(-\text{scale})$ is the precision).
Data_Representation : enumeration (Array, Boolean, Character, Enum, Float, Fixed, Integer, String, Struct, Union) applies to (data); The <code>Data_Representation</code> property may be used to specify the representation of simple or composite data types within the programming language source code. Note: An implementation is allowed to support only a subset of these types.
Dimension : list of aadlinteger applies to (data); The <code>Dimension</code> property is used to specify the dimensions of a multi-dimensional array, the i^{th} value in the list representing the i^{th} dimension of the array. This property shall be used in conjunction with the <code>Data_Representation</code> property.
Element_Names : list of aadlstring applies to (data); The <code>Element_Names</code> provides the names of a struct or union members in order of appearance as defined by the <code>Base_Type</code> property.
Enumerators : list of aadlstring applies to (data); The <code>Enumerators</code> provides the list of enumeration literals attached to an enumeration data component.
IEEE754_Precision : enumeration (Simple, Double) applies to (data); The <code>IEEE754_Precision</code> property indicates, for a float data component type, the precision used. This property is derived from the notion of precision per the 754-1985 IEEE Standard for Binary Floating-Point Arithmetic.
Initial_Value : list of aadlstring applies to (data, port, parameter); <code>Initial_Value</code> specifies a list of initial values for a data component or port in string form. For a subprogram parameter, it defines a default value. It can be used to represent initial values other than strings as string. This (list of) string is interpreted by the source language processor. In this case, the core AADL language processor does not check consistency of the initial values with the data type of the data component.
Integer_Range : range of aadlinteger applies to (data, port, parameter); <code>Integer_Range</code> specifies a range of integer values that apply to the data component. This property is used to represent integer range constraints on data that is of some integer type.
Measurement_Unit : aadlstring applies to (data, port, parameter); The <code>Measurement_Unit</code> property specifies the measurement unit of the data being communicated. A full list of recommended name for units, as part of the SI standard is proposed by the International Bureau of Weight and Measures: http://www.bipm.org/en/si/.



Note: this should not be mixed with AADL units, which specify units to be applied to properties, e.g. thread's period, bandwidth usage that belong solely to the architecture modeling

Number_Representation : **enumeration** (Signed, Unsigned)

applies to (data);

Number_Representation specifies whether an integer data component is signed or unsigned.

Real_Range: **range of aadlreal**

applies to (data, port, parameter);

Real_Range specifies a range of real values that apply to the data component. This property is used to represent real range constraints on data that is of some real type.

Representation : **list of aadlstring**

applies to (data);

Representation specified the actual representation of enumerator's value.

end Data_Model;

Annex B.4 Predeclared AADL Package for Basic Data Types

- (1) This AADL package defines a set of basic types for use as data component types in AADL models. Typically, they can be used to represent the base type of a data component or port.

Note: These data types are different from the built in property types such as **aadlinteger**, **aadlreal**, **aadlboolean**, or **aadlstring**. The latter denote types manipulated to represent entities of the AADL model, whereas the types defined below represent types manipulated by the system being modeled.

- (2) This package belongs to the set of models defined by the AADL V2. Implementations of AADL tools are allowed to add additional properties to clarify some elements of a data component (e.g. project specific limits) or to help model processing.
- (3) Implementations are not allowed to modify the values of the properties defined for the types defined below.
- (4) A method of implementation may restrict the types from this package that are supported, e.g. to generate code, to reflect implementation limits.

package Base_Types

public

data Boolean

properties

Data_Model::Data_Representation => Boolean;

end Boolean;

data Integer

properties

Data_Model::Data_Representation => Integer;

end Integer;

-- Signed integer of various byte sizes

data Integer_8 **extends** Integer

properties

```
Data_Model::Number_Representation => Signed;  
Source_Data_Size => 1 Bytes;  
end Integer_8;
```

```
data Integer_16 extends Integer
```

properties

```
Data_Model::Number_Representation => Signed;  
Source_Data_Size => 2 Bytes;  
end Integer_16;
```

```
data Integer_32 extends Integer
```

properties

```
Data_Model::Number_Representation => Signed;  
Source_Data_Size => 4 Bytes;  
end Integer_32;
```

```
data Integer_64 extends Integer
```

properties

```
Data_Model::Number_Representation => Signed;  
Source_Data_Size => 8 Bytes;  
end Integer_64;
```

```
-- Unsigned integer of various byte sizes
```

```
data Unsigned_8 extends Integer
```

properties

```
Data_Model::Number_Representation => Unsigned;  
Source_Data_Size => 1 Bytes;  
end Unsigned_8;
```

```
data Unsigned_16 extends Integer
```

properties

```
Data_Model::Number_Representation => Unsigned;  
Source_Data_Size => 2 Bytes;  
end Unsigned_16;
```

```
data Unsigned_32 extends Integer
```

properties

```
Data_Model::Number_Representation => Unsigned;
Source_Data_Size => 4 Bytes;
end Unsigned_32;

data Unsigned_64 extends Integer
properties
  Data_Model::Number_Representation => Unsigned;
  Source_Data_Size => 8 Bytes;
end Unsigned_64;

data Natural extends Integer
properties
  Data_Model::Integer_Range => 0 .. Max_Target_Integer;
end Natural;

data Float
properties
  Data_Model::Data_Representation => Float;
end Float;

data Float_32 extends Float
properties
  Data_Model::IEEE754_Precision => Simple;
  Source_Data_Size => 4 Bytes;
end Float_32;

data Float_64 extends Float
properties
  Data_Model::IEEE754_Precision => Double;
  Source_Data_Size => 8 Bytes;
end Float_64;

data Character
properties
  Data_Model::Data_Representation => Character;
end Character;

data String
properties
  Data_Model::Data_Representation => String;
end String;
end Base_Types;
```

Annex B.5 Examples

(1) This AADL package lists some examples of the AADL modeling property sets.

```
package Base_Types::Example_Types
public
  with Data_Model, Base_Types;

  data One_Dimension_Array
  properties
    Data_Model::Data_Representation => Array;
    Data_Model::Base_Type => (classifier (Base_Types::Integer));
    Data_Model::Dimension => (42);
  end One_Dimension_Array;

  data Two_Dimensions_Array
  properties
    Data_Model::Data_Representation => Array;
    Data_Model::Base_Type => (classifier (Base_Types::Integer));
    Data_Model::Dimension => (74, 75);
  end Two_Dimensions_Array;

  -- Two equivalent ways to define a structure with named
  -- elements

  data A_Struct1
  properties
    Data_Model::Data_Representation => Struct;
    Data_Model::Base_Type => (classifier (Base_Types::Float),
                             classifier (Base_Types::Character));
    Data_Model::Element_Names => ("f1", "c2");
  end A_Struct1;

  data A_Struct2
  properties
    Data_Model::Data_Representation => Struct;
  end A_Struct2;

  data implementation A_Struct2.impl
  subcomponents
    f1 : data Base_Types::Float;
    c2 : data Base_Types::Character;
  end A_Struct2.impl;
```

```
-- Two equivalent ways to define a union with named elements
```

```
data A_Union1
properties
  Data_Model::Data_Representation => Union;
  Data_Model::Base_Type => (classifier (Base_Types::Float),
                           classifier (Base_Types::Character));
  Data_Model::Element_Names => ("f1", "f2");
end A_Union1;
```

```
data A_Union2
properties
  Data_Model::Data_Representation => Union;
end A_Union2;
```

```
data implementation A_Union2.impl
subcomponents
  f1 : data Base_Types::Float;
  c2 : data Base_Types::Character;
end A_Union2.impl;
```

```
data An_Enum
properties
  Data_Model::Data_Representation => Enum;
  Data_Model::Enumerators => ("foo", "bar");
  Data_Model::Representation => ("00", "11");
end An_Enum;
```

```
data A_Fixed_Point
properties
  Data_Model::Data_Representation => Fixed;
  Data_Model::Data_Digits => 18;
  Data_Model::Data_Scale => 8;
end A_Fixed_Point;
```

```
data A_Kanji extends Base_types::Character
properties
  Data_Model::Code_Set => 16#30001#;
```

```
-- ISO code for Japanese characters
end A_Kanji;

data Russian_Strings extends Base_Types::String
properties
  Data_Model::Code_Set => 16#15#;
  -- ISO code for Cyrillic characters
end Russian_Strings;

data Bounded_Strings extends Base_Types::String
properties
  Data_Model::Dimension => (10);
end Bounded_Strings;

-- Representation of a type defined in another modeling
-- notation, or programming language

data ASN1_Type
properties
  Source_Language => ASN1;
  Source_Text => ("ASN1_types.asn");
  Type_Source_Name => "the_type";
end ASN1_Type;

data C_Type
properties
  Source_Language => C;
  Source_Text => ("types.h");
  Type_Source_Name => "the_type";
end C_Type;

end Base_Types::Example_Types;
```

Annex Document D Behavior Model

Normative

Annex D.1 Scope

- (1) The Behavior Annex document provides a standard sublanguage extension to allow behavior specifications to be attached to AADL components. The aim of the Behavior Annex is to refine the implicit behavior specifications that are specified by the core of the language. The Behavior Annex targets the following goals:
 - Describe the internal behavior of component implementations as a state transition system with guards and actions. However, the aim is not to replace software programming languages or to express complex subprogram computations.
 - Extend the default run-time execution semantics that is specified by the core of the standard, such as thread dispatch protocols.
 - Provide more precise subprogram calls synchronization protocols for client-server architectures.
- (2) The Behavior Annex document includes the definition of the Behavior_Specification AADL annex subclause that is used to describe internal behavior and run-time semantics extensions and the Behavior_Properties property set that defines a property for subprogram call protocols. This document is organized as follows:
 - Annex D.2 gives a short overview of the annex
 - Annex D.3 defines the syntax and semantics of the **state automaton** used for the Behavior_Specification annex subclause.
 - Annex D.4 defines the syntax and semantics of the **dispatch condition language** used to specify the conditions for a thread dispatch that are a refinement of the default thread dispatch conditions specified in the core AADL standard.
 - Annex D.5 defines the syntax and semantics of the **interaction operations** used to specify the component interaction with other components through its port, subprogram, and data access features.
 - Annex D.6 defines the syntax and semantics of the **action language** used to specify the transition actions of the automaton.
 - Annex D.7 defines the syntax and semantics of the **expression language** used in the various parts of the behavior annex.
 - Annex D.8 provides details about the behavior of subprogram call **synchronization protocols** defined by the Subprogram_Call_Protocol property.

Annex D.2 Overview of Behavior Annex Concepts

- (1) The Behavior_Specification annex is expressed as state transition systems with guards and actions. The behaviors described in this annex must be seen as specifications of the actual behaviors: they can therefore be non-deterministic. They are based on state variables whose evolution is specified by transitions that can be characterized by conditions and actions. The action language can make use of all the visible declarations that are described in the encompassing AADL specification. When such an annex is defined in the scope of a component type declaration, then it applies as a default behavior to all the implementations of that type.

- (2) The state transition system of a Behavior_Specification consists of a collection of states and transitions between the states. The state automaton has one initial state, from which the automaton behavior starts. The state automaton also has one (or more) final states. When this state is reached the behavior is considered to have completed. The state automaton can have complete states that represent temporary suspension of execution and resumption based on external trigger conditions. Finally, the state automaton can have discrete states of execution behavior.
- (3) These state transition systems can be used to specify the sequential execution behavior of an AADL subprogram, the dispatch, mode, input, and output behavior of AADL threads or devices, the protocol behavior of AADL virtual buses and buses, the dynamic behavior of a process or system, etc. The behavior from the initial state to a final state typically represents the execution behavior of a subprogram with one or more return points. The behavior of a component such as a sampling periodic thread or a thread processing commands, may start in an initial state; initialize itself and suspend itself at a complete state; reactivate from the complete state repeatedly based on time or the arrival of an external event or message; transitioning from the initial state to the first complete state, or between complete states may involve transitioning to intermediate computational states; a termination request results in a transition to a final state.
- (4) The transitions of a state transition system specify behavior as a change of the current state from a source state to a destination state. A condition determines whether a transition is taken, and an action is performed when the condition evaluates to true. Transition priorities control the evaluation order to transition conditions, thus, the transition to be taken if the conditions of multiple transitions hold; otherwise the transition choice is non-deterministic. Transition conditions fall into two categories: conditions that affect the execution of a thread based on external triggers (dispatch conditions); and conditions that model behavior within an execution sequence of a thread, subprogram or other component (execution conditions) and are based on input values from ports, shared data, parameters, and behavior variable values.
- (5) The timed dispatch protocol of the core AADL standard specifies a time out condition for dispatches relative to the previous dispatch. The Behavior_Specification allows the modeler to define dispatch time out conditions relative to the previous completion as well as time out conditions on blocks of behavior actions. Furthermore, the Behavior_Specification allows modelers to specify what behavior is to occur when such time outs occur.
- (6) When a component is specified in the core AADL model to have modes, then the Behavior_Specification supports the specification of mode-specific behavior.
- (7) A subprogram call is an external trigger to the state transition system of a subprogram. The arrival of events and event data on ports of a non periodic thread is an external trigger for dispatching the thread; it initiates a transition in the hybrid state automaton defined in the AADL core standard [AS5506A 5.4.1] as well as the state transition system of the Behavior_Specification of the thread. The transmission request on an outgoing port is an external trigger to the state transition system of a virtual bus or bus. Dispatch conditions are specified in terms of external triggers via event port, event data port, calls received on provides subprogram access features, or time out. Dispatch does not depend on the input value, which can only impact the action following the dispatch.
- (8) External trigger conditions, consumption of input and generation of output must be consistent with the input/output semantics of the core AADL standard. For example, if the core model specifies a subset of the ports to be external dispatch triggers, then the external trigger condition can only specify conditions on this subset. Similarly, additional ports can be specified in both the core model and in the Behavior_Specification to indicate that the port content is frozen at dispatch time. These two specifications must be consistent.
- (9) Input on ports is frozen according to the semantics of the core AADL standard [AS5506A 8.3.2] and made available to the application Behavior_Specification in the form of a port variable. Newly arriving data, events, and event data do not affect the content of the port variable. In the case of a data port the current value at input freeze time is made available. In the case of event ports and event data ports the port queue content is handled according to the specified dequeuing protocol. One, several, or all dequeued elements are made available to the Behavior_Specification.
- (10) The Behavior_Specification of a component may access a shared data component made accessible through a *requires data access* feature. The current data value of such shared data component available to the Behavior_Specification is determined at the time of access.

Annex D.3 Behavior Specification

- (1) Behavior specifications can be attached to any AADL component types and component implementations using an annex subclause. When defined within component type specifications, it represents behavior common to all the associated implementations. If a component type or implementation is extended, behavior annex subclause defined in the ancestor are applied to the descendent except if the later defines its own behavior annex subclause.
- (2) The Behavior Annex supports behavior specification for components with modes in two ways. First, a behavior annex subclause can be specified for a specific mode by declaring it with an **in modes** statement [AS5506A 12]. If mode specific behavior annex subclauses are defined, those modes for which no behavior annex is given have an empty behavior specification. If no mode specific behavior annex subclause is defined, then it applies for all modes.
- (3) Second, modes can be reflected in a Behavior_Specification without **in modes**. In this case a mode can be reflected by a complete state of the same name in the Behavior_Specification and mode transition behavior can be modeled as a transition out of such a complete state whose condition identifies the event port named in the mode transition, if specified in the core AADL model.
- (4) The behavior annex defines a transition system (an extended automaton) described by three sections:
 - variables declarations;
 - states declarations;
 - transitions declarations;
- (5) The variables section declares identifiers that represent variables with the scope of the behavior annex subclause in which they are declared.
- (6) Local variables can be used to keep track of intermediate results within the scope of the annex subclause. They may hold the values of out parameters on subprogram calls to be made available as parameter values to other calls, as output through enclosing out parameters and ports, or as value to be written to a data component in the AADL specification. They can also be used to hold input from incoming port queues or values read from data components in the AADL specification. They are not persistent across the various invocations of the same behavior annex subclause.
- (7) State variables can be used to reduce the size of the state automaton by keeping track of counts. Such state variables must be declared in the core specification as data subcomponents or requires data access features. The same visibility rules as those defined in the core standard apply to the behavior annex subclause, for instance, a behavior annex subclause defined in a component type declaration cannot reference a data subcomponent as a state variable.
- (8) The states section declares all the states of the automaton. Some states may be qualified as *initial* state, *final* state, or *complete* state, or combinations thereof. A state without qualification will be referred to as execution state. A behavior automaton starts from an initial state and terminates in a final state.
- (9) In the case of subprograms, the automaton consists of one *initial* state representing the starting point of a call, zero or more intermediate execution states, and one *final* state. A *final* state represents the completion of a call. The *complete* state is not used in behavior specifications of subprograms.
- (10) In the case of threads and devices, the automaton consists of one *initial* state representing the state before initialization actions, one or more *complete* states, zero or more intermediate execution states, and one or several *final* states. A *complete* state acts as a suspend/resume state out of which threads and devices are dispatched. The *final* state represents the state when a thread or device completes finalization. In the case of threads, these qualified states have equivalents in the hybrid automata that define the runtime system behavior in the core AADL standard [AS5506A 5.4.1]. Note that one state can play the role of *initial*, *complete*, and *final* state.
- (11) Other components, such as processes, have behavior automata with an *initial* state, one or more *complete* states and one *final* state. The *initial* state represents the state before initialization begins. The *final* state represents the state after finalization completes.
- (12) A state that is qualified as *final*, and is not at the same time *initial* or *complete*, cannot accept outgoing transitions. If the purpose of the behavior annex is to provide a specification of the intended behavior of a component, then the use of several *final* states is allowed. Otherwise, if the purpose is to provide a deterministic representation of the implementation of the internal behavior of the component, then only one *final* state must be defined.

- (13) When a subprogram has modes it may have a separate behavior annex subclause for each mode. In this case, a mode transition results in a transition to the *initial* state of the behavior automaton of the new mode at the next subprogram call.
- (14) When a component other than a subprogram has modes it may also have a separate behavior annex subclause for each mode. In this case, a mode transition results in a transition from the *complete* state of the current mode behavior automaton to the *initial* state of the behavior automaton of the new mode.
- (15) The transitions section defines transitions from a source state to a destination state. Transitions in a subprogram behavior automaton represent the control flow sequence through execution states to a *final* state. Transitions in a thread behavior automaton represent the execution sequence within a thread. A transition out of a *complete* state is initiated by a dispatch once the dispatch condition is satisfied.
- (16) Transitions can be guarded by dispatch conditions, or execute conditions.
- (17) Dispatch conditions explicitly specify dispatch trigger conditions out of a *complete* state. If not specified, the default dispatch conditions provided by the core AADL standard apply. A dispatch condition is a Boolean expression that specifies the logical combination of triggering events for the next dispatch. A dispatch triggering event can be the arrival of an event or event data on an event port or an event data port, the receipt of a call on a provided subprogram access, or a time out event. The ports listed in the dispatch condition must be consistent with the ports listed in the core AADL model as dispatch triggers.
- (18) Execute conditions specify transition conditions out of an execution state to another state. They can also be used to express initialization sequence when applied on a transition out of a state that is qualified as *initial* and is not at the same time *complete* or *final*. They effectively select between multiple transitions out of a given state to different states. These conditions are logical expressions based on component inputs, subcomponent outputs, and values of data components, state variable values, and property constants. They can also result in catching a previously raised execution Timeout exception.
- (19) If transitions have been assigned a priority number, then the priority determines the transition to be taken. The higher the priority number is the higher the priority of the transition is. If more than one transition out of a state evaluates its condition to true and no priority is specified, then one transition is chosen non-deterministically. For multiple transitions with the same priority value the selection is also non-deterministic. Transitions with no specified priority have the lowest priority.
- (20) Each transition can have actions. Actions can be subprogram calls, retrieval of input and sending of output, assignments to variables, read/write to data components, and time consuming activities. An action is related to the transition and not to the states: if a transition is taken, the sequence of actions is performed and then the state specified as the destination of the transition becomes the new current state.

Semantics

- (21) A behavior annex subclause is used to specify behaviors of components, such as the interaction behavior with other components, discrete state behavior of the component itself including time consuming actions, functional behavior based on external conditions, protocol behavior of virtual buses and buses, logical control flow in terms of subprogram calls within a thread or a subprogram. These behavior specifications must be consistent with the respective semantics in the core AADL standard.
- (22) A behavior annex subclause may be interpreted as a refinement of a call sequence section in a thread or subprogram component implementation. If both a call sequence section and a behavior annex subclause with subprogram call actions are defined for the same component implementation, then all the subprogram calls specified in the former must be reflected in the latter, although the call order may differ.
- (23) The core AADL standard defines runtime execution states for threads. These states include an *initial* state (thread halted), a *complete* state (awaiting dispatch,) and a *final* state (stopped thread). The behavior annex specification for a thread must be consistent with the core AADL semantics.

- (24) The behavior specification of components other than subprograms consists of an *initial* state, one or more *final* states, one or more *complete* states, and zero or more execution states. A transition out of the *initial* state is triggered by the *initialize* action defined in the core AADL standard. Execution states may be used to represent intermediate initialization steps. Upon completion of initialization a *complete* state is reached. A dispatch trigger will result in a transition out of a *complete* state to zero or one execution state. A dispatch trigger can be the arrival of input on ports, a subprogram call initiated by another thread, or a timed event (periodic dispatch or timeout). Reaching a *complete* state can be interpreted as calling the `Await_Dispatch` run-time service. Thus a component is suspended if it performs a transition to a *complete* state, after having executed the action associated to the transition. The next dispatch will restart the thread from that state.
- (25) The core AADL standard defines dispatch conditions for threads in terms of a disjunction of trigger conditions as result of arrival of events or event data on incoming ports of subprogram access features. A subset of ports involved in the triggering of a dispatch may be specified through the `Dispatch_Trigger` property. The behavior specification can refine this dispatch condition into a Boolean condition that is associated with a transition out of a *complete* state.
- (26) The dispatch conditions are evaluated to determine whether a dispatch occurs. If there are multiple outgoing transitions the execution condition (if present) is evaluated to determine which transition is taken. If multiple transitions are eligible then the priority value determines an evaluation ordering, otherwise one of the eligible transitions is taken non-deterministically. The higher the priority value is, the higher the priority of the transition is.
- (27) When the `Dispatch_Protocol` property is *timed* or *hybrid*, the value of the time out dispatch condition is given by the `Period` property of the component.
- (28) Local variables are temporary placeholders for application data that may not require temporary storage. For example, they are used to indicate which out parameter of a subprogram call is used as in parameter of another call or as output in a port variable. Local variables must be explicitly typed with a valid data component classifier.
- (29) An *otherwise* execute condition becomes true when all the other execute conditions associated with transitions from the same state are false.

Syntax

```

behavior_annex ::=
  [ variables { behavior_variable }+ ]
  [ states { behavior_state }+ ]
  [ transitions { behavior_transition }+ ]

behavior_variable ::=
  local_variable_declarator { , local_variable_declarator }*
  : data_unique_component_classifier_reference;

unique_component_classifier_reference ::=
  { package_identifier :: }* component_type_identifier
  [ . component_implementation_identifier ]

behavior_state ::=
  behavior_state_identifier { , behavior_state_identifier }*
  : behavior_state_kind state;

behavior_state_kind ::=
  [ initial ] [ complete ] [ final ]

behavior_transition ::=
  [ transition_identifier [ [ behavior_transition_priority ] ] : ]
  source_state_identifier { , source_state_identifier }*
  -[ behavior_condition ]->
  destination_state_identifier [ behavior_action_block ] ;

```

```

behavior_transition_priority ::=
    numeral

behavior_condition ::=
    dispatch_condition
  | execute_condition

execute_condition ::=
    [ logical_value_expression | behavior_action_block_timeout_catch | otherwise ]

timeout_catch ::=
    timeout

declarator ::=
    identifier { array_size } *

array_size ::
    [ integer_value_constant ]

```

Naming Rules

- (N1) The variable, state and transition identifiers must be unique within a behavior annex subclause. In addition, they must not conflict with mode identifiers, feature identifiers, or data subcomponent identifiers of the component for which the behavior annex subclause is defined. An exception to this rule are identifiers for complete states that represent modes.
- (N2) An empty execute condition is equivalent to a condition that is always true.

Legality Rules

- (L1) A behavior annex specification for a subprogram must define an initial state and one final state.
- (L2) A behavior annex specification for a subprogram must not define any complete states.
- (L3) A behavior annex specification for a thread, device, and other components that can be suspended awaiting dispatch or awaiting a mode transition, must define at least one complete state and one initial state. This may be the same state.
- (L4) A behavior annex specification for threads and other components with initialization and finalization entrypoints may explicitly model the initialization and finalization by including one initial state and one or more final states.
- (L5) A behavior annex specification for a subprogram must not contain a dispatch condition in any of its transitions.
- (L6) Only transitions out of complete states may have dispatch conditions.
- (L7) Transitions out of complete states must have dispatch conditions.
- (L8) Transitions from states that are final only are not allowed.
- (L9) The specification of frozen ports in the dispatch condition must be consistent with that of the core AADL model [AS5506A 5.4.8].

Consistency Rules

- (C1) If a component type or implementation is extended, behavior annex subclause defined in the ancestor are applied to the descendent except if the later defines its own behavior annex subclause.
- (C2) A behavior annex subclause of a subcomponent overrides the behavior annex subclause of its containing component if they conflict

- (C3) The behavior annex state transition system must not remain blocked in an execution state. This means that the logical disjunction of all the execute conditions associated with the transitions out of an execution state must be true.
- (C4) If the behavior annex defines transitions from a complete state that represents a mode in the containing component, then the `behavior_condition` associated with these transitions must be consistent with the corresponding `mode_transition_triggers` [AS5506A 12]

Examples

-- This first example shows the use of variable labels as local variables to
 -- pass data between successive actions.

```
package BA_example1 public

data number
end number;

subprogram mul
features
  x : in parameter number;
  y : in parameter number;
  z : out parameter number;
end mul;

subprogram cube
features
  x : in parameter number;
  y : out parameter number;
  mul : requires subprogram access mul;
end cube;

subprogram implementation cube.ba
annex behavior_specification {**
  variables tmp : number;
  states s : initial final state;
  transitions t : s -[]-> s { mul!(x,x,tmp); mul!(tmp,x,y) };
**};
end cube.ba;
```

-- In the preceding code, `mul!` represents a call to subprogram `mul`. This can
 -- thus be used as a shortcut for the following equivalent call specification
 -- expressed in core AADL:

```
subprogram implementation cube.no_ba
calls t : {
  mul1 : subprogram mul;
  mul2 : subprogram mul;
};
connections
  parameter x -> mul1.x;
  parameter x -> mul1.y;
  parameter mul1.z -> mul2.x;
  parameter x -> mul2.y;
  parameter mul2.z -> y;
end cube.no_ba;

end BA_example1;
```

-- Next example exploits the AADL execution model to compute a speed by
 -- counting the number of ticks received each second by the measuring thread.