AEROSPACE STANDARD

AS5506™ REV. D

Draft 2019-05

Superceding AS5506C

Architecture Analysis and Design Language (AADL): Introduction

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.

TABLE OF CONTENTS

No table of figures entries found.

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 standard addresses the requirements defined in SAE ARD 5296, Requirements for the Avionics Architecture Description Language.

(3) The AADL standard consists of a core language standard that is defined in this document and a collection of standardized property sets and/or sublanguages that are defined in annex documents. The core language standard provides full support for modeling the application task and communication architecture, the hardware platform, and the physical environment of embedded software-intensive systems, including standardized predeclared properties to characterize task execution and communication timing, as well as deployment of the application on the hardware platform. The standardized extensions allow core AADL models to be annotated with information that is not represented by the core language to meet specific embedded system analysis needs such as security analysis, dependability analysis, and behavioral analysis, and support for automated generation and integration of systems.

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.

(4) 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.

(5) This document contains a revised version of the AADL core language, that enables modeling of systems in terms of interacting components, defines the semantics for specific software and platform component and component categories, and provides a property language and type system to define and associate properties with AADL model elements.

INFORMATION AND FEEDBACK

(1) The website at http://www.aadl.info is an information source regarding the SAE AADL standard and its use. The standard document itself is available at https://www.sae.org by searching for AS5506.

(2) The website http://osate.org provides provides information and a download site for the Open Source AADL Tool Environment (OSATE).

(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

1. SCOPE

(1) This standard defines a language for describing both the software architecture and the execution platform architectures of performance-critical, embedded, real-time systems; the language is known as the SAE Architecture Analysis & Design Language (AADL). An AADL model describes a system as a hierarchy of components with their interfaces and their interconnections. Properties are associated to these constructions. AADL components fall into two major categories: those that represent the physical hardware and those representing the application software. The former is typified by processors, buses, memory, and devices, the latter by application software functions, data, threads, and processes. The model describes how these components interact and are integrated to form complete systems. It describes both functional interfaces and aspects critical for performance of individual components and assemblies of components. The changes to the runtime architecture are modeled as operational modes and mode transitions.

(2) The language is applicable to systems that are real-time, resource-constrained, safety-critical systems, and those that may include specialized device hardware.

(3) This standard defines the core AADL that is designed to be extensible. While the core language provides a number of modeling concepts with precise semantics including the mapping to execution platforms and the specification of execution time behavior, it is not possible to foresee all possible architecture analyses. Extensions to accommodate new analyses and unique hardware attributes take the form of new properties and analysis specific notations that can be associated with components. Users or tool vendors may define these extensions. Extensions may be proposed as annex documents for inclusion in the AADL standard.

(4) This standard does not specify how the detailed design or implementation details of software and hardware components are to be specified. Those details can be specified by a variety of software programming and hardware description languages. The standard specifies relevant characteristics of the detailed design and implementation descriptions, such as source text written in a programming language or hardware description language, from an external (black box) perspective. These relevant characteristics are specified as AADL component properties, and as rules of conformance between the properties and the described components.

(5) This standard does not prescribe any particular system integration technologies, such as operating system or middleware application program interfaces or bus technologies or topologies. However, specific system architecture topologies, such as the ARINC 653 executives, can be modeled through software and execution platform components. AADL can be used to describe a variety of hardware architectures and software infrastructures. Integration technologies can be used to implement a specified system. The standard specifies rules of conformance between AADL system architecture specifications and actual system implementations.

(6) The standard was not designed around a particular set of tools. It is anticipated that systems and software tools will be provided to support the use of AADL.

1.1 Purpose

(1) The purpose of AADL is to provide a standard and sufficiently precise (machine-processable) way of modeling the architecture of an embedded, real-time system, such as an avionics system or automotive control system, to permit analysis of its properties, and to support the predictable integration of its implementation. Defining a standard way to describe system components, interfaces, and assemblies of components facilitates the exchange of engineering data between the multiple organizations and technical disciplines that are invariably involved in an embedded real-time system development effort. A precise and machine-processable way to describe conceptual and runtime architectures provides a framework for system modeling and analysis; facilitates the automation of code generation, system build, and other development activities; and significantly reduces design and implementation defects.

(2) AADL describes application software and execution platform components of a system, and the way in which components are assembled to form a complete system or subsystem. The language addresses the needs of system developers in that it can describe common functional (control and data flow) interfacing idioms as well as performance-critical aspects relating to timing, resource allocation, fault-tolerance, safety and certification.

(3) AADL describes functional interfaces and non-functional properties of application software and execution platform components. The language is not suited for detailed design or implementation of components. AADL may be used in conjunction with existing standard languages in these areas. AADL describes interfaces and properties of execution platform components including processor, memory, communication channels, and devices interfacing with

the external environment. Detailed designs for such hardware components may be specified by associating source text written in a hardware description language such as VHDL. AADL can describe interfaces and properties of application software components implemented in source text, such as threads, processes, and runtime configurations. Detailed designs and implementations of algorithms for such components may be specified by associating source text written in a software programming language such as Ada or C, or domain-specific modeling languages such as MatLab®/Simulink® .

(4) AADL describes how components are composed together and how they interact to form complete system architectures. Runtime semantics of these components are specified in this standard. Various mechanisms are available to exchange control and data between components, including message passing, event passing, synchronized access to shared components, and remote procedure calls. Thread scheduling protocols and timing requirements may be specified. Dynamic reconfiguration of the runtime architecture may be specified through operational modes and mode transitions. The language does not require the use of any specific hardware architecture or any specific runtime software infrastructure.

(5) Rules of conformance are specified between specifications written in AADL, source text and physical components described by those specifications, and physical systems constructed from those specifications. The AADL is not intended to describe all possible aspects of any possible component or system; selected syntactic and semantic requirements are imposed on components and systems. Many of the attributes of an AADL component are represented in an AADL model as properties of that component. The conformance rules of the language include the characteristics described by these properties as well as the syntactic and semantic requirements imposed on components and systems. Compliance between AADL specifications and items described by specifications is determined through analysis, e.g., by tools for source text processing and system integration.

(6) AADL can be used for multiple activities in multiple development phases, beginning with preliminary system design. The language can be used by multiple tools to automate various levels of modeling, analysis, implementation, integration, verification and certification.

1.2 Field of Application

(1) AADL was developed to model embedded systems that have challenging resource (size, weight, power) constraints and strict real-time response requirements. Such systems should tolerate faults and may utilize specialized hardware such as I/O devices. These systems are often certified to high levels of assurance. Intended fields of application include avionics systems, automotive systems, flight management systems, engine and power train control systems, medical devices, industrial process control equipment, robotics, and space applications. AADL may be extended to support other applications as the need arises.

1.3 Structure of Document

1.3.1 A Reader's Guide

(1) As necessary, the term AADL V3 will be used to refer to the revised version of AADL defined in this document. The AADL standard suite consists of this core language document and a set of annex documents of standardized extensions. This core language document contains a number of sections and appendices. The sections define the core AADL. The appendices provide additional information, both normative and informative about the core language. Annex documents introduce additional standardized properties and possibly language extensions in the form of specialized notations.

(2) The core AADL standard document consists several parts. This Introductory Part consists of the following:

- Section 2, References, provides normative and applicable references as well as terms and definitions.
- Section 3, Architecture Analysis & Design Language Summary, introduces and defines the concepts of the language.
- Section 4, Lexical Elements

(3) Part 1 defines a set of general concepts of AADL as an architecture description language. They are

- Packages to organize the specification of systems as AADL models

- Interfaces, implementations and configurations to represent specifications of components.and subcomponents as their instantiation,
- Features to describe external interaction points of components,
- Connections and feature delegation to specify interaction between components,
- Flow specifications and sequences to describe flows throught components as abstraction and as realization,
- Modes to support modeling of operational modes with mode-specific system configurations and property values.

(4) Part 2 introduces specific categories of components and defines their static semantics for modeling application and execution platform components in modeled systems or systems of systems. The component categories are:

- Application system software components are thread, thread group, process, data, subprogram, subprogram group,.
- Execution platform components are processor, virtual processor, memory and virtual memory, bus, virtual bus, and device.
- System as a compositional modeling element that combines execution platform and application system software components.
- Abstract component as a generic component without specific sementic meaning.

(5) Part 3 defines the dymanic semantics of execution and communication specified by an AADL model.

(6) Part 4 defines the concept of properties through property definitions, assignment of property values. It also defines a type system that is used for proeprties as well as AADL model elements. Finally it defines an expression language to constrain and evaluate AADL models.

(7) The document also includes a number of Appendix sections.

- Appendix 1 contains the standard AADL set of predeclared properties, property constants, and data types.
- Appendix 2 contains a glossary of terms.
- Appendix 3 contains a summary of the syntax as defined in the sections of this document.

(8) The annex documents introduce additions and extensions to the core AADL.

- The Runtime System and Code Generation Annex provides guidance for automatic generation and integration of runtime systems and application code in different implementation languages. It defines a standardized set of properties for recording mappings from the AADL model to source text and for automatic code generation.
- The Data Modeling Annex provides guidance on data modeling and how to map relevant data modeling information into an AADL model if desirable. It defines a standardized set of properties and basic data types in support of data modeling.
- The Error Model Annex defines a standardized core language extension in the form of a sublanguage notation and properties the component to support annotating AADL models with safety-criticality and dependability related information of a system.
- The Behavior Annex defines a standardized core language extension in the form of a sublanguage notation to specify the behavior of AADL components as AADL model annotations.

(9) The core language and the annex documents are *normative*, except that the material in each of the items listed below is informative:

- Text under a NOTE or Examples heading.
- The Examples segement.

(10) All implementations shall conform to the core language. In addition, an implementation may conform separately to one or more Annexes that represent extensions to the core language.

(11) The following appendices and annexes are informative and do not form a part of the formal specification of AADL:

- xxx

1.3.2 Structure of Document Sections

(1) Each section of the core standard is organized into the following segments:

- *Description* describes the static and dynamic semantics of different AADL constructs with respect to the system they model. The semantics are concerned with the effects of the execution of the constructs, not how they would be specifically executed in a computational tool.
- *Syntax* describes syntax rules, concerned with the organization of the symbols in the AADL expressions, are given in Extended Backus-Naur-Form (EBNF).
- *Naming rules* define scoping of defining names and resolution of references.
- *Legality rules* define semantic restrictions on AADL specifications. Legality rules must be validated by AADL processing tools when a model is loaded into the tool.
- *Consistency rules* define consistency restrictions on system instances. A consistency rule must be validated by AADL processing tools upon a user request or when an analysis method that relies on the rule is invoked.
- *Standard properties* lists the properties that are defined for the concept described in a given section.
- *Processing Requirements and Permissions* documents additional requirements and permissions for determining compliance. Providers of processing method implementations must document a list of those capabilities they support and those they do not support.
- *Examples* illustrate the possible forms of the constructs described. This material is informative.

(2) All paragraphs are numbered with numbering restarting with each section. Naming rules, legality rules, and consistency rules have their own paragraph numbering also restarting with each section. They can be identified by section number and paragraph number.

1.4 Compliance

(1) An AADL specification is compliant with the AADL core language standard if it satisfies all the syntactic and legality rules. An AADL specification is compliant with an AADL Annex standard if it satisfies all the syntactic and legality rules defined in the respective normative Annex.

(2) A component or system is *compliant* with an AADL specification of that component or system if the nominal and exceptional behaviors of that component or system satisfy the applicable semantics of the AADL specification, as defined by the semantic rules in this standard. A component or system may be a physical implementation (e.g., a piece of hardware), or may be a model (e.g., a simulation or analytic model). A model component or system may exhibit only partial semantics (e.g., a schedulability model only exhibits temporal semantics). Physical components and systems must exhibit all specified semantics, except as permitted by this standard.

(3) *Noncompliance* of a component with its specification is a kind of design fault. This may be handled by run-time exception handling and fault-tolerance in an implemented actual system. A developer is permitted to classify such components as anomalous rather than noncompliant.

(4) A tool that operates on AADL specifications is *compliant* with the core language standard if the tool checks for compliance of input specifications with the syntactic and legality rules defined herein, except where explicit permission is given to omit a check; and if all physical or model components or systems generated by the tool are compliant with the specifications used to generate those components or systems. The AADL standard allows profiles of language subsets to be defined and requires a minimum subset of the language to be supported. A tool must clearly specify any portion of the language not supported and warn the user if a specification contains unsupported language constructs, when appropriate. A tool is compliant with the XMI interchange format if it supports saving and reading of AADL model in the XMI interchange format. A tool is compliant with an annex if the tool checks for compliance of input specifications with the syntactic and legality rules defined in the respective annex document.

(5) Compliance of an AADL specification with the syntactic and legality rules can be automatically checked, with the exception of a few legality rules that are not in general tractably checkable for all specifications. Compliance of a component or system with its specification, and compliance of a tool with this standard, cannot in general be fully

automatically checked. A verification process that assures compliance to the degree required for a particular purpose must be used to perform the latter two kinds of compliance checking.

1.5 Method of Description and Syntax Notation

(1) The language is described by means of a context-free syntax together with context-dependent requirements expressed by narrative rules. The meaning of a construct in the language is defined by means of narrative rules.

(2) The context-free syntax of the language is described using the variant Backus-Naur Form (BNF) [BNF 1960] as defined herein.

- Lower case words in `courier new` font, some containing embedded underlines, are used to denote syntactic categories. A syntactic category is a nonterminal in the grammar. For example:

```
component_feature_list
```

- Boldface words are used to denote reserved words, for example:

```
implementation
```

- A vertical line separates alternative items.

```
software_category ::= thread | process
```

- Square brackets enclose optional items. Thus the two following rules are equivalent.

```
property_association ::= property_name => [ constant ] expression

property_association ::=

     property_name => expression

  | property_name => constant expression
```

- Curly brackets with a * symbol enclose a repeated item. The item may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule. Thus the two following rules are equivalent.

```
declaration_list ::= declaration { declaration }*

declaration_list ::= declaration

                   | declaration declaration_list
```

- Curly brackets with a + symbol specify a repeated item with one or more occurrences. Thus the two following rules are equivalent.

```
declaration_list ::= { declaration }+

declaration_list ::= declaration { declaration }*
```

- Parentheses (round brackets) enclose several items to group terms. This capability reduces the number of extra rules to be introduced. Thus, the first rule is equivalent with the latter two.

```
property_association ::= identifier ( =>| +=>) property_expression

property_association ::= identifier assign property_expression
```

```
assign ::= => | +=>
```

- Square brackets, curly brackets, and parentheses may appear as delimiters in the language as well as meta-characters in the grammar. Square, curly, and parentheses that are delimiters in the language will be written in bold face in grammar rules, for example:

```
property_association_list ::=

    { property_association { ; property_association }*  }
```

- The syntax rules may preface the name of a nonterminal with an italicized name to add semantic information. These italicized prefaces are to be treated as comments and not a part of the grammar definition. Thus the two following rules are equivalent.

```
component ::= identifier : component_classifier ;

component ::= component_identifier : component_classifier ;
```

- A construct is a piece of text (explicit or implicit) that is an instance of a syntactic category, for example:

```
My_GPS: thread GPS.dualmode ;
```

(3) The syntax description has been developed with an emphasis on an abstract syntax representation to provide clarity to the reader.

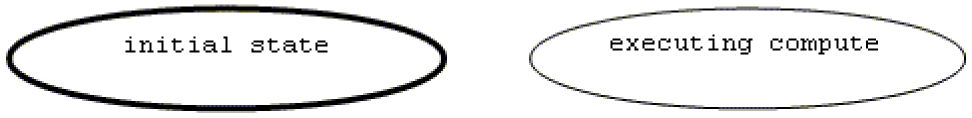
1.6 Method of Description for Discrete and Temporal Semantics

(1) Discrete and temporal semantics of the language are defined in sections that define AADL concepts using a concurrent hierarchical hybrid automata notation, together with additional narrative rules about those diagrams. This notation consists of a hierarchical finite state machine notation, augmented with real-valued variables to denote time and time-varying values, and with edge guard and state invariant predicates over those variables to define temporal constraints on when discrete state transitions may occur.

(2) A semantic diagram defines the nominal scheduling and reconfiguration behavior for a modeled system as well as scheduling and reconfiguration behavior when failures are detected. A physical realization of a specification may violate this definition, for example due to runtime errors. A violation of the defined semantics is called an anomalous behavior. Certain kinds of anomalous behaviors are permitted by this standard. Legal anomalous behaviors are defined in the narrative rules.

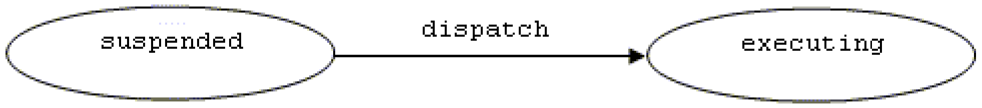
(3) Semantics for individual components are defined using a sequential hierarchical hybrid automaton. System semantics are defined as the concurrent composition of the hybrid automata of the system components.

(4) Ovals labeled with lower case phrases are used to denote discrete states. A component may remain in one of its discrete states for an interval of time whose duration may be zero or greater. Every semantic automaton for a component has a unique initial discrete state, indicated by a heavy border. For example,



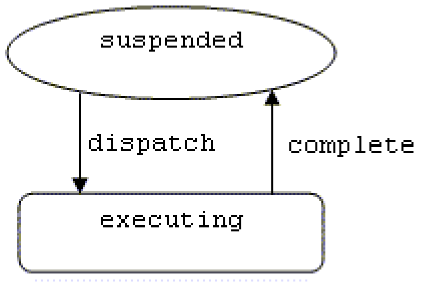
(5) Directed edges labeled with one or more comma-separated, lower case phrases are used to denote possible transitions between the discrete states of a component. Transitions over an edge are logically instantaneous, i.e.,

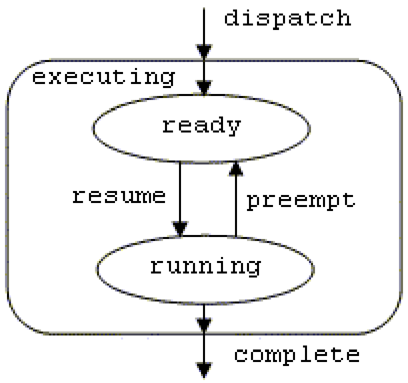
the time interval in which a transition from a discrete state (called the source discrete state) to a discrete state (called the destination discrete state) has duration 0. For example,



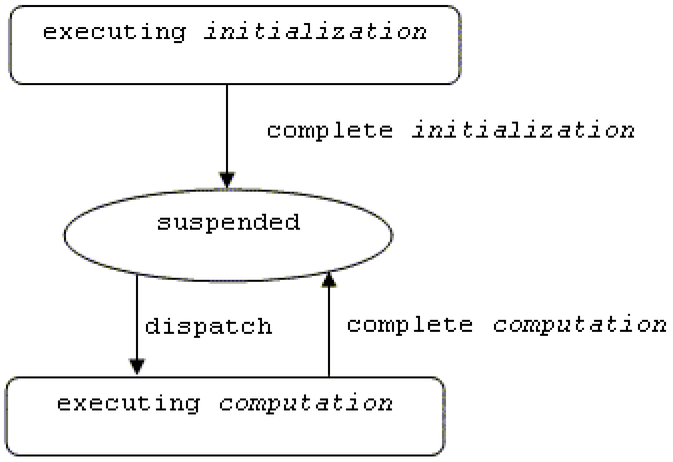
(6) Permissions that allow a runtime implementation of a transition to occur over an interval of time are expressed as narrative rules. However, all implemented transitions must be atomic with respect to each other, all observable serializations must be admitted by the logical semantics, and all temporal predicates as defined in subsequent paragraphs must be satisfied.

(7) Hybrid automaton can have hierarchical states. Oblong boxes labeled with lower case phrases denote abstract discrete states, for which another hybrid semantics diagram with an identically labeled oblong box for which another hybrid semantics diagram with an identically labeled oblong box specifies the discrete states and edges that make up that abstract discrete state. For example,



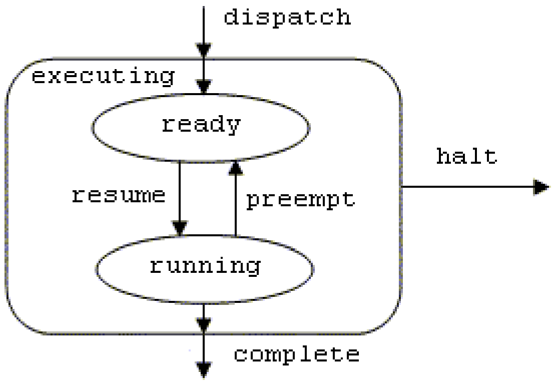


(8) Abstract discrete states are reusable, i.e., a hybrid semantics diagram can contain several oblong boxes with the same label An abstract state label or an edge label may include italicized letters that are not a part of the formal name but are used to distinguish multiple instances. For example, both abstract discrete states below will be defined by a single diagram labeled `executing`.



(9) If there is an external edge that enters or exits an abstract discrete state in the defining diagram for, and there are no edges within that definition that connect any internal discrete state with that external edge, then there implicitly exist edges from every contained discrete state in the defining diagram to or from that external edge. In that case, a transition into or out of an abstract discrete state represents transitions into or out any of its internal states. For

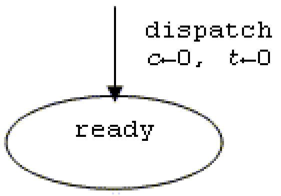
example, in the following diagram there is an implicitly defined `halt` edge out of both the `ready` and the `running` discrete states.



(10) Real-valued variables whose values are time-varying may appear in expressions that annotate discrete states and edges of hybrid semantic diagrams. Specific forms of annotation are defined in subsequent paragraphs. The set of real-valued variables associated with a semantic diagram are those that appear in any expression in that diagram, or in any of the defining diagrams for abstract discrete states that appear in that diagram. Real-valued time-varying variables will be named using an italicized front. The initial values for the real-valued time-varying variables of a hybrid semantic diagram are undefined whenever they are not explicitly defined in narrative rules.

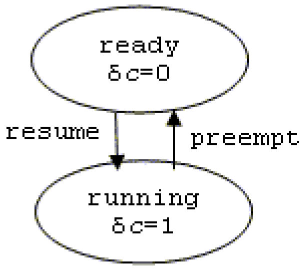
(11) In addition to standard rational literals and arithmetic operators, expressions may also contain functions of discrete variables. The names of functions and discrete variables will begin with upper case letters. The semantics for function symbols and discrete variables will be defined using narrative rules. For example, the subexpression `Max(Compute Execution Time)` may appear in a semantic diagram, together with a narrative rule stating that the value is the maximum value of a range-valued component property named `Compute Execution Time`.

(12) Edges may be annotated with assignments of values to variables associated with the semantic diagram. When a transition occurs over an edge, the values of the variables are set to the assigned values. For example, in the following diagram, the values of the variables *c* and *t* are set to 0 when the component transitions into the `ready` discrete state.

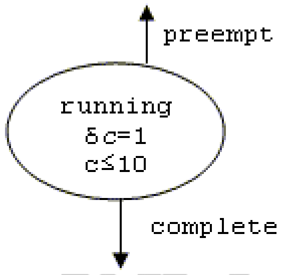


(13) Discrete states may be annotated with expressions that define the possible rates of change for real-valued variables during the duration of time a component is in that discrete state. The rate of a variable is denoted using the symbol δ, for example δ*x*=[0,1] (the rate of the variable *x* may be any real value in the range of 0 to 1). If, rates of change are not explicitly shown within a discrete state for a time-varying variable, then the rate of change of that variable in that state is defined to be 1. For example, in the following diagram the rate of change for the variable *c* is 1 while the

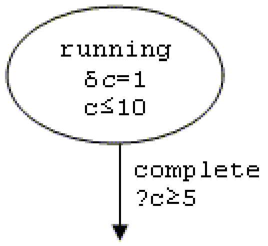
component is in the discrete state `running`, but its value remains fixed while the component is in the `ready` state, equal to the value that existed when the component transitioned into the `ready` state.



(14) A discrete state may be annotated with Boolean-valued expressions called invariants of that discrete state. In this standard, all semantic diagrams are defined so that the values of the variables will always satisfy the invariants of a discrete state for every possible transition into that discrete state. A transition must occur out of a discrete state before the values of any time-varying variables cause any invariant of that discrete state to become false. Invariants are used to define bounds on the duration of time that a component can remain in a discrete state. For example, in the following diagram the component must transition out of the `running` state before the value of the variable *c* exceeds 10.



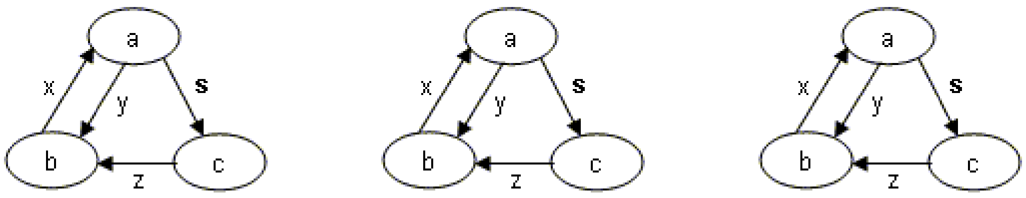
(15) An edge may be annotated with Boolean-valued expressions called guards of that edge. A transition may occur from a source discrete state to a destination discrete state only when the values of the variables satisfy all guards for an edge between those discrete states. A guard on an edge is evaluated before any assignments on that edge are performed. For example, in the following diagram the component may only `complete` when the value of the variable *c* is 5 or greater (but must `complete` before *c* exceeds 10 because of the invariant).



(16) A sequential semantic automaton defines semantics for a single component. A system may contain multiple components. The semantics of a system are defined to be the concurrent composition of the sequential semantic automata for each component. Except as described below, every component is represented by a copy of its defined semantic automaton. All discrete states and labels, all edges and labels, and all variables, are local to a component. The set of discrete states of the system is the cross-product of the sets of discrete states for each of its cross product components. The set of transitions that may occur for a system at any point in time is the union of the transitions that may occur at that instant for any of its components.

(17) If an edge label appears in boldface, then a transition may occur over that edge only when a transition occurs over all edges having that same boldface label within the synchronization scope for that label. The synchronization scope for a boldface label is indicated in parentheses. For example, if a transition occurs over an edge having a boldface label with a synchronization scope of process, then every thread contained in that process in which that boldface label appears anywhere in its hybrid semantic diagram must transition over some edge having that label. That is, transitions over edges with boldface labels occur synchronously with all similarly labeled edge transitions in all

components associated with the component with the specified synchronization scope as described in the narrative. Furthermore, every component in that synchronization scope that might participate in such a transition in any of its discrete states must be in one of those discrete states and participate in that transition. For example, when the synchronization scope for the edge label **s** is the same for all three of the following concurrent semantic automata, a transition over the edge labeled **s** may only occur when all three components are in their discrete states labeled `a`, and all three components simultaneously transition to their discrete states labeled `c`.



(18) If a variable appears in boldface, then there is a single instance of that variable that is shared by all components in the synchronization scope of the variable. The synchronization scope for a boldface variable will be defined in narrative rules.

2. REFERENCES

2.1 SAE Publications

(1) 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.

(2) SAE AS-5506C:2012, Architecture Analysis & Design Language (AADL), Jan 2017 [AS5506C].

(3) SAE AS-5506/1A:2006, Architecture Analysis & Design Language (AADL) Annex Volume 1, Sept 2015.

(4) SAE AS-5506/2:2011, Architecture Analysis & Design Language (AADL) Annex Volume 2, Jan 2011.

2.2 IEEE Publications

(1) Available from IEEE, 445 Hoes Lane, Piscataway, NJ 08854-1331, Tel: 732-981-0060, www.ieee.org.

(2) IEEE Standard 1003.1-2001 Information Technology - Portable Operating System Interface (POSIX). Institute of Electrical and Electronic Engineers [IEEE 2001].

(3) IEEE Standard 1003.13-1998 Information Technology - Standardized Application Environment Profile - POSIX Realtime Application Support (AEP). The Institute of Electrical and Electronics Engineers [IEEE 1998].

(4) IEEE Standard 1003.5b-1996 Information Technology - POSIX Ada Language Interfaces - Part 1: Binding for System Application Program Interface (API) - Amendment 1: Realtime Extensions. The Institute of Electrical and Engineering Electronics [IEEE 1996].

2.3 ISO Publications

(1) Available from ISO, 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, Tel: +41 22 749 01 11, www.iso.org.

(5) ISO/IEC 8652:1995 (E) International Organization for Standardization, Information Technology - Programming Languages - Ada [ISO 1995]

(6) ISO/IEC 8652:1995/COR.1:2001 International Organization for Standardization, Technical Corrigendum to Information Technology - Programming Languages - Ada [ISO 2001]

(7) ISO/IEC 9899:1999 International Organization for Standardization, Information Technology - Programming Languages - C [ISO 1999]

(8) ISO/IEC TR 15942:2000 International Standards Organization, Guide for the Use of the Ada Programming Language in High Integrity Systems [ISO 2000]

(9) ISO/IEC/IEEE 24765:2010 Systems and software engineering — Vocabulary, Dec 2010.

2.4 Terms and Definitions

(1) Terms are introduced throughout this standard, indicated by italic type. Informational definitions of terms are given in Appendix B , Glossary. Definitions of terms used from other standards, such as ISO/IEC/IEEE 24765 Systems and Software Engineering – Vocabulary, Dec. 2010, ISO/IEC 9945-1:1996 [IEEE/ANSI Std 1003.1, 1996 Edition], Information Technology – Portable Operating System Interface (POSIX), or IFIP WG10.4 Dependability: Basic Concepts and Terminology [IFIP WG10.4-1992], are so marked. Terms not defined in this standard are to be interpreted according to the Webster's Third New International Dictionary of the English Language. Terms explicitly defined in this standard are not to be presumed to refer implicitly to similar terms defined elsewhere. A full description of the syntax and semantics of the concept represented by the terms is found in the respective document sections, clauses, and subclauses.

3. ARCHITECTURE ANALYSIS & DESIGN LANGUAGE SUMMARY

(1) This section provides an informative overview of AADL concepts, structure, and use. In this section the first appearance of a term that has a specific meaning in this standard will be italicized.

(2) An AADL specification represents a component model of a computer system runtime architecture that consists of the application software (typically embedded, safety-critical, mission-critical, or performance-critical) expressed in terms of interacting tasks, and the execution platform, i.e., the logical and physical computing hardware, as well as the interface to the physical system. A component represents a part of a system and interacts with other components. A system is hierarchically composed of interacting components with interface enforcement at every level.

(3) A *package* provides a library-like structure for organizing component classifiers, data types, property definitions, and annex specific definitions into separate namespaces, similar to Java packages that are used to organize Java class declarations. Packages can have a nested naming hierarchy. A component classifier in a package is referenced by qualifying its name with the package name. Package elements that have been declared as private are not visible outside a package. The content of a package is made visible inside another package through a *import* statement. In addition package elements can always be referenced by qualifying them with the package name.

(4) *Classifiers* are used to describe components in terms of their interface, implementation and configuration.

(5) An *interface* represents a specification of a component in terms of features for interaction with other components, binding points for platform/resource deployment binding, flow specifications from component inputs to component outputs, modes as externally visible operational states, and property values to characterize the component. Implementations of the component are required to satisfy this specification. Interfaces can be *compositional*, i.e., interfaces can be defined in terms of features defined in other interfaces and as containing *named interfaces*, i.e., named collections of features.

(6) A *feature* describes an interaction point through which control and data may be provided to or required from other components. Features can be ports to support directional flow of control and data, subprogram access to represent remote procedure calls, data access to represent coordinated access to shared data, and bus access as well as virtual bus access to represent platform component interactions via buses and virtual buses.

(7) An *implementation* specifies a realization (blue print) of the component in terms of subcomponents, connections between the features of those subcomponents, flows across a sequence of subcomponents, modes, and implementation specific property values. Bindings specify the binding of a software subcomponent to a platform component. AADL allows multiple implementations to associated with an interface to represent component variants.

(8) A *subcomponent* declares a component instance that is contained in another component by naming its classifier. The component hierarchy of a system instance is determined by recursively instantiating the subcomponents of a top-level system.

(9) A *configuration* configures subcomponents, including nested subcomponents, with classifiers, annotates model elements with property values, and adds binding specifications, flows, and annex subclauses. Once configured those aspects cannot be changed. Component configurations are compositional in that multiple configurations can be assigned to the same component as long as they do not conflict, e.g., security properties and bindings may be defined in separate configurations and combined to complete a system specification for analysis. A parameterized configuration definitions limits the user to only configures those aspects of a system that are specified by parameters.

(10) *Properties* are used to represent attributes and other characteristics, such as the period and deadline of threads. Any namable model element, e.g., components, features, modes, connections, flows, and subprogram calls, can have properties. When properties are associated with declarations of component interfaces, implementations, configurations, features, subcomponents, connections, flows, modes and binding points, they apply to all respective model element instances within an instance model. This standard defines a set of predeclared properties. For

example, a predeclared property is used to specify the period of a thread. Users can define additional properties to support new forms of system analysis.

(11) *Data types* are used to define the value type of properties, the type of data for ports and data components, as well as the type of variables in annex sublanguages. This standard defines a set of predeclared data types and measurement units. Users can define additional application specific data types.

(12) AADL support the specification of partial models such as models of the application software only, the hardware only, specification of a top-level architecture in terms of its subsystems without their realizations, and specification of component templates. Classifier extension declarations can specify new component interfaces andimplementations by adding features, subcomponents, connections, flows, and properties to previously defined interfaces and implementations. In addition component configurations refine existing interfaces and implementations. This allows conceptual and reference architectures to be specified and to be refined into fully specified runtime architectures, and partial system specifications to evolve into fully specified and configured systems including the deployment of application software on the computing hardware

(13) *Component categories* associate specific semantics to components. The following component categories are defined as part of the AADL standard: thread, thread group, process, data, subprogram, subprogram group as *application software* component categories, memory, virtual memory, bus, virtual bus, processor, virtual processor, and device *as execution platform* component categories, and system and abstract as general compositional components. They form the core of the AADL modeling vocabulary.

(14) Application software components represent concurrent tasks and their interactions, protected address spaces, and source text,. Source text can be written in a programming language such as Ada, C, or Java, or domain-specific modeling languages such as Simulink, SDL, ESTEREL, LUSTRE, and UML, for which executable code may be generated. The source text modeled by a software component may represent a partial application program or model (e.g., they form one or more independent compilation units as defined by the applicable programming language standard). Rules and permissions governing the mapping between AADL specification and source text depend on the applicable programming or modeling language standard. Predeclared component properties identify the source text container and the mapping of AADL concepts to source text declarations and statements. These properties also specify memory and execution times requirements and other known characteristics of the component.

(15) *Thread* components model concurrent tasks or active objects, i.e., concurrent logical threads. Each logical thread represents an execution sequence through source text (or more exactly, through binary images produced from the compilation, linking and loading of source text). A scheduler manages the execution of a thread. Logical threads may be executed by separate operating system (OS) threads, or they may be combined into a single operating system thread. The dynamic semantics for a thread are defined in this standard using hybrid automata. The threads can be in states such as suspended, ready, and running. State transitions occur as a result of dispatch requests, faults, and runtime service calls. They can also occur if time constraints are exceeded. Error detection and recovery semantics are specified. Dispatch semantics are given for standard dispatch protocols such as periodic, sporadic, aperiodic, timed, hybrid, and background threads. Additional dispatch protocols may be defined. Threads can contain subprogram and data components, and provide or require access to data components.

(16) *Thread groups* support structural grouping of threads within a process. A thread group may contain data, thread, and thread group subcomponents. A thread group may require and provide access to data components.

(17) *Process* components model space partitions in terms of virtual address spaces containing source text that forms complete programs as defined in the applicable programming language standard. 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 binary image produced by compiling and linking this source text must execute properly when loaded into a unique

virtual address space. As processes do not represent concurrent tasks, they must contain at least one thread. Processes can contain thread groups, threads, and data components, and can access or share data components.

(18) *Data* components represent static data. These data components can be accessed by one or more threads and processes; they do so by indicating that they require access to the external data component. Concurrent access to data is managed by the appropriate concurrency control protocol as specified by a property.

(19) *Subprogram* components model source text that is executed sequentially. Subprograms are callable from within threads and subprograms. Subprograms may require access to data components and may contain data subcomponents to represent local variables. *Subprogram groups* represent source code libraries.

(20) Execution platform components represent computing hardware components that are capable of scheduling threads, of enforcing specified address space protection at runtime, of storing source text code and data and of performing communication for application system connections. The device component represents elements of the physical environment that an embedded system interacts with, such as sensors, actuators, or engines.

(21) *Processor* components are an abstraction of hardware and software that is responsible for scheduling and executing threads. In other words, a processor may include functionality provided by operating systems. Alternatively, operating systems can be modeled like application components. Processors can contain memory and require access to buses. Processors can support different scheduling protocols. Threads are bound to processors for scheduling and execution.

(22) *Virtual processor* components represent virtual machines or hierarchical schedulers. Threads can be bound to them. Virtual processors can be used in two ways. Processors and virtual processors can be subdivided into virtual processors by declaring virtual processor subcomponents. Virtual processors can also be declared separately and explicitly bound to processors and virtual processors.

(23) *Memory* components model randomly accessible physical storage such as RAM or ROM. Memories have properties such as the number and size of addressable storage locations. Binary images of source text are bound to memory. Memory can contain nested memory components. Memory components require access to buses.

(24) *Virtual memory* components model logical address spaces.

(25) *Bus* components model communication channels that can exchange control and data between processors, memories, and devices. A bus is typically hardware that supports specific communication protocols, possibly implemented through software. Processors, memories, and devices communicate by accessing a shared bus. Buses can be directly connected to other buses. Logical connections between threads that are bound to different processors transmit their information across buses that provide the physical connection between the processors. Buses can require access to other buses.

(26) *Virtual bus* components represent virtual channels or communication protocols that perform transmission within processors or across buses. Virtual buses can be subcomponents of buses and virtual buses, or virtual buses can be declared separately and explicitly bound to buses and virtual buses.

(27) *Device* components model 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 of the modeled system or its (simulated) software equivalent. Examples of devices are timers, which exhibit simple behavior, or a camera or GPS, which exhibit complex behavior. Devices are logically connected to application software components and physically connected to processors via buses. They cannot store nor execute external application software source text themselves, but may include driver software executed on a connected processor. A device requires access to buses.

(28) *System* components model hierarchical compositions of software and execution platform components. A system may directly contain data, subprogram, subprogram groups, process, memory, processor, virtual processor, bus, virtual bus, device, system as well as abstract subcomponents. Thread and thread group subcomponents must be declared in processes and are indirectly part of a system that contains these processes. A system component may require and provide access to data and bus components. Execution platform component can be system components in their own right and be modeled using system implementations. For example, a system implementation can be associated with a device that models a camera. This system implementation describes the internal of the camera in

terms of the CCD sensor a device, a DSP processor, a general purpose processor as well a software that implements the image processing and download capability of the camera.

(29) *Abstract* components represent generic components such as components in a functional architecture.

(30) *Modes* represent the operational states of software, execution platform, and compositional components in the modeled physical system. A component can have mode-specific property values. A component can also have mode-specific configurations of different subsets of subcomponents and connections. In other words, a mode change can change the set of active components and connections. Mode transitions model dynamic operational behavior that represents switching between configurations and changes in component-internal characteristics, such as conditional execution source text sequences or operational states of a device, that are reflected in property values. Other examples of mode-specific property values include the period or the worst-case execution time of a thread. A change in operating mode can have the effect of activating and deactivating threads for execution and changing the pattern of connections between threads. A mode subclause in a component implementation specifies the mode states and mode change behavior in terms of transitions; it specifies the events as transition triggers. Subcomponent and connection declarations as well as property associations declare their applicability (participation) in specific modes.

(31) *Connections* specify interaction between components at runtime. Connections are declared between features of subcomponents. In addition features of an enclosing component can be *delegated* to features of subcomponents. Named interfaces allow groups of features to be connected through a single connection declaration. A connection instance on a system is represented by a connection declaration and zero or more delegate declarations that follow the component hierarchy to the ultimate connection source and the ultimate connection destination. For example, there is a connection declaration within System1 from Process3's out port to Process4's in port. The source port of the connection is delegated to an out port of Thread1. The destination port of the connection is delegated to an in port of Thread2.

(32) *Flow specifications* describe externally observable flow of information through a component. Flow specifications represent flow sources, i.e., flows originating from within a component, flow sinks, i.e., flows ending within a component, and flow paths, i.e., flows through a component from its incoming ports to its outgoing ports.

(33) *Flow sequences* describe actual flow representing flow specifications as sequences of subcomponent flows and connections. They are declared in component implementations. A flow sequence may also represent an *end-to-end flow* that starts within one subcomponent and ends within another subcomponent.

(34) An actual embedded system is represented by an instance of an AADL component implementation or configuration that consists of subcomponents representing the application software, the computing platform, and the physical environment.

(35) An AADL specification may be used in a variety of ways by a variety of tools during a broad range of life-cycle activities, e.g., for documentation during preliminary specification, for schedulability or reliability analysis during design studies and during verification, for generation of system integration code during implementation. Note that application software components must be bound to execution platform components - ultimately threads to processors and binary images to memory in order for the system to be analyzable for runtime properties and the actual system to be constructed from the AADL specification. Many uses of an AADL specification need not be fully automated, e.g., some implementation steps may be performed by hand.

(36) The AADL core language is extensible through collections of data type and property definitions, annex subclauses and annex libraries that can be standardized or user-defined.

(37) *Annex subclauses* consist of specifications in an annex-specific sublanguage that can be added to classifiers. *Annex libraries* are packages that contain reusable definitions expressed in an annex-specific sublanguage.

(38) Standardized Annexes are the Error Model Annex, Behavior Annex, Data Modeling Annex, ARINC653 Annex. The Error Model Annex provides a notation for adding error behavior annotations for safety and security related analyses. The Behavior Annex provides a notation for component and component interaction behavior specifications. The Data Modeling Annex provides guidance on how to approach data modeling with AADL. The ARINC653 Annex provides

guidance and properties for modeling embedded software system architectures that comply with the ARINC653 standard.

(39) This standard defines basic concepts and requirements for determining compliance between a component specification via classifier and an actual component. This standard does not restrict the lower-level representation(s) used for software components, e.g., binary images, conventional programming languages, application modeling languages, nor does it restrict the lower-level representation(s) used for physical hardware component designs, e.g., circuit diagrams,hardware behavioral descriptions.

4. GENERAL SYNTAX AND LEXICAL ELEMENTS

(1) AADL is a case sensitive language.

(2) Identifiers consist of upper case and lower case letters, digits, and the underscore character. Reserved words must be lower case.

(3) The text of an AADL description consists of a sequence of lexical elements, separators, and delimiters. The rules of composition are given in this section.

4.1 Lexical Elements, Separators, and Delimiters

Description

(1) The text of an AADL specification consists of a sequence of separate lexical elements. Each lexical element is formed from a sequence of characters. It is either a delimiter, an identifier, a reserved word, a literal, or a comment. The meaning of an AADL specification depends only on the particular sequences of lexical elements that form its compilations, excluding comments.

(2) In some cases an explicit separator is required to separate adjacent lexical elements. A separator is any of a space character, a tabulation character, or the end of a line.

(3) One or more separators are allowed between any two adjacent lexical elements, before the first, or after the last. At least one separator is required between an identifier, a reserved word, or a numeric literal and an adjacent identifier, reserved word, or numeric literal.

(4) A *delimiter* is either one of the following special characters

( ) [ ] { } , . : ; = * + -

(5) or one of the following *compound delimiters* each composed of two or three adjacent special characters

:: => -> <-> .. –[ ]–> {* *}

(6) Each of the special characters listed for single character delimiters is a single delimiter except if that character is used as a character of a compound delimiter, or as a character of a literal, or comment.

Implementation Permissions

(7) An implementation may limit the supported character set to the ASCII character set and symbols used by delimiters and separators.

4.2 Identifiers

Description

(1) Identifiers consist of a letter followed by a sequence of letters or digits optionally separated by an underscore character. Identifiers are case sensitive. Identifiers are used in names.

Syntax

```
identifier ::= letter { [ underscore ] letter_or_digit }*

letter_or_digit ::= letter | digit

letter ::=
  ‘a’ | ‘b’ | ‘c’ | ‘d’ | ‘e’ | ‘f’ | ‘g’  ‘h’ | ‘i’ | ‘j’ | ‘k’ | ‘l’ | ‘m’ | ‘n’
  | ‘o’ | ‘p’ | ‘q’ | ‘r’ | ‘s’ | ‘t’ | ‘u’ | ‘v’ | ‘w’ | ‘x’ | ‘y’ | ‘z’
  | ‘A’ | ‘B’ | ‘C’ | ‘D’ | ‘E’ | ‘F’ | ‘G’  ‘H’ | ‘I’ | ‘J’ | ‘K’ | ‘L’ | ‘M’ | ‘N’
  | ‘O’ | ‘P’ | ‘Q’ | ‘R’ | ‘S’ | ‘T’ | ‘U’ | ‘V’ | ‘W’ | ‘X’ | ‘Y’ | ‘Z’
```

```
digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

underscore :== '_'
```

Legality Rules

(L1) An indentifier must be distinct from reserved words.

4.3 Numerical Literals

Description

(1) There are two kinds of numeric literals, real and integer. A real_literal is a numeric_literal that includes a point; an integer_literal is a numeric_literal without a point.

Syntax

```
numeric_literal ::= integer_literal | real_literal | based_integer_literal
```

4.3.1 Decimal Integer Literals

Description

(1) A decimal integer literal is a numeric_literal in the conventional decimal notation (that is, the base is ten). An underscore character in a numeral does not affect its meaning.

Syntax

```
decimal_integer_literal ::= numeral [ positive_exponent ]

numeral ::= digit { [ underscore ] digit }*

exponent ::= ( 'E' | 'e' ) [+] numeral | ( 'E' | 'e' ) – numeral

positive_exponent ::= ( 'E' | 'e' ) [+] numeral
```

Examples

```
     120           1E6          123_456 -- integer literals
     12.0          0.0          0.456        3.14159_26     -- real literals
```

4.3.2 Decimal Real Literals

Description

(1) A decimal real literal is a numeric_literal in the conventional decimal notation (that is, the base is ten). An underscore character in a numeral does not affect its meaning.

Syntax

```
decimal_real_literal ::= numeral . numeral [ exponent ]
```

Examples

```
     120           1E6          123_456 -- integer literals
     12.0          0.0          0.456        3.14159_26     -- real literals
```

4.3.3 Based Integer Literals

Description

(1) A based integer literal is an integer literal expressed in base binary, octal, or hexadecimal. An underscore character in a numeral does not affect its meaning.

Syntax

```
based_integer_literal ::= bin_integer | oct_integer | hex_integer


bin_integer ::= '0' ( 'b' | 'B' } bin_digit { [ underscore ] bin_digit }


oct_integer ::= '0' ( 'o' | 'O' } oct_digit { [ underscore ] oct_digit }


hex_integer ::= '0' ( 'x' | 'X' } hex_digit { [ underscore ] hex_digit }


hex_digit ::=
  digit | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f'


oct_digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7'


bin_digit ::= '0' | '1'
```

Examples

```
     0b1111_1111       0xFF    -- integer literals of value 255
     0B1110_0000       0o340   -- integer literals of value 224
```

4.4 String Literals

Description

(1) A string literal is formed by a sequence of graphic characters (possibly none) enclosed between two quotation marks.

(2) A null string literal is a string_literal with no string_elements between the quotation marks.

NOTE: An end of line cannot appear in a string_literal.

Syntax

```
string_literal ::= "{string_element}*"
string_element ::= "" | non_quotation_mark_graphic_character
```

Legality Rules

(L2) A non_quotation_mark_graphic_character is any character of ISO 10646 BMP that is not a control function and is not a quotation mark.

4.5 Comments

Description

(1) A comment starts with two adjacent hyphens and extends up to the end of the line.

(2) The presence or absence of comments has no influence on whether a program is legal or illegal. Furthermore, comments do not influence the meaning of a program; their sole purpose is the enlightenment of the human reader.

Syntax

```
comment ::= --{non_end_of_line_character}*
```

Examples

```
--  this is a comment

end;  --  processing of Line is complete

--  a long comment may be split onto
--  two or more consecutive lines

----------------  the first two hyphens start the comment
```

4.6 Reserved Words

(1) Reserved words in AADL are all lower case.

(2) The following are the AADL reserved words:

```
abstract, access, all, and, annex, as, binding, boolean, bus,
connection, data, device, end, enumeration, event, extends,
false, feature, flow, for, group, import, interface, in, initial, integer, is,
list, memory, mode, not, or, out, package, parameter, path, port, private,
process, processor, property, properties, provides, range, real, record, reverse,
requires, sink, source, string, subprogram, system, thread, true,
type, units, use, virtual
```

NOTE: the list of reserved words needs to be updated.