



AEROSPACE STANDARD

AS5506™/3

Issued 2017-08
Reaffirmed 2025-10

Superseding AS5506/2 Annex D

Architecture Analysis and Design Language (AADL) Annex D: Behavior Model Annex

RATIONALE

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

The purpose of the Behavior Model Annex is to enable modeling of component and component interaction behavior in a state-machine based annex sublanguage. The Behavior Model Annex language was originally published as AS5506/2 in 2011. The language addresses a number of errata and improvements to align with the AADL V2.2 core language AS5506C published January 2017. These errata and changes have been approved by the committee.

The Behavior Model Annex was originally published as part of AS5506/2, a volume of three annexes. In the future, each published Annex to AADL has its own number starting with the revised Behavior Model Annex as AS5506/3.

AS5506/3 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_3

For more information on this standard, visit
<https://www.sae.org/standards/content/AS5506/3/>

Documents

Standard Sharing Hub

TABLE OF CONTENTS

Annex D	Behavior Model	6
Annex D.1	Scope	6
Annex D.2	Overview Of Behavior Annex Concepts.....	6
Annex D.3	Behavior Specification.....	8
Annex D.4	Thread Dispatch Behavior Specification.....	15
Annex D.5	Component Interaction Behavior Specification.....	19
Annex D.6	Behavior Action Language	23
Annex D.7	Behavior Expression Language	29
Annex D.8	Synchronization Protocols	34
Figure 1	Sender behavior specification.....	18
Figure 2	Client server behavior	36

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 a Behavior Model Annex that extends the AADL core language [AS5506C] with a state machine-based sublanguage annex notation for specifying component and component interaction behavior.

INTRODUCTION

- (1) The SAE Architecture Analysis and 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 contains a revised version of the Behavior Annex, that enables modeling of component and component interaction behavior in a state-machine based annex sublanguage.

INFORMATION AND FEEDBACK

- (1) The website at <http://www.aadl.info> is an information source regarding the SAE AADL standard. It makes available papers on AADL, its benefits, and its use. Also available are papers on MetaH, the technology that demonstrated the practicality of a model-based system engineering approach based on architecture description languages for embedded real-time systems.
- (2) The website provides links to three SAE AADL related discussion forums:
 - The SAE AADL User Forum to ask questions and share experiences about modeling with SAE AADL,
 - The AADL Toolset User Forum to ask questions and share experiences with the Open Source AADL Tool Environment (OSATE), and
 - The SAE Standard Document Corrections and Improvements Forum that records errata, corrections, and improvements to the current release of the SAE AADL standard.
- (3) 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.
- (4) Questions and inquiries regarding working versions of annexes and future versions of the standard can be addressed to info@aadl.info.
- (5) 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
- (6) 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.
- (7) Multiple comments per e-mail message are acceptable. Please use a descriptive "Subject" in your e-mail message.
- (8) 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

Annex D - Behavior Model

Normative

Annex D.1 Scope

- (1) This Behavior Annex 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 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 sublanguage.
 - 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 sublanguage expresses 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 a *final* state is reached the behavior is considered to have terminated. 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 transitory *execution* states.
- (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 *execution* 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 of 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 four categories:
- conditions that affect the execution of a thread based on external triggers (dispatch conditions);
 - conditions that model behavior within an execution sequence of a thread, subprogram or other component (execute conditions) and are based on input values from ports, shared data, parameters, and behavior variable values;
 - conditions that abstract the behavior of a component when detecting internal events;
 - conditions that abstract the behavior of a component when receiving external events that do not represent dispatch conditions (e.g., mode switches).
- (5) The *timed* dispatch protocol of the core AADL standard specifies a timeout condition for dispatches relative to the previous dispatch. The Behavior_Specification allows the modeler to define dispatch timeout conditions relative to the previous completion as well as timeout conditions on blocks of behavior actions. Furthermore, the Behavior_Specification allows modelers to specify what behavior is to occur when such timeouts 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 an *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 [AS5506C, 5.4.1] as well as the state transition system of the Behavior_Specification of the thread. The transmission request on an outgoing port may be 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 timeout. 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, which must be consistent with the specification based on the core AADL standard.
- (9) Input on ports is frozen according to the semantics of the core AADL standard [AS5506C, 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 values of such shared data components are determined at the time of access.

Annex D.3 Behavior Specification

- (1) Behavior specifications can be attached to any AADL component types and implementations using *Behavior_Specification* annex subclauses. 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_Specification* annex subclause can be specified for a specific mode by declaring it with an *in modes* statement [AS5506C, Section 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_Specification* 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 local variables with the scope of the *Behavior_Specification* 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_Specification* 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_Specification* annex subclause defined in a component type declaration cannot reference a data subcomponent of an implementation of that type as a state variable.

- (8) The **states** section declares the states of the automaton. A state 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 or more *final* states. 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. A *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 [AS5506C, 5.4.1]. Note that one state can play the role of *initial*, *complete*, and *final* state.
- (11) Other components, such as processes, may 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_Specification 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_Specification 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, execute conditions, external conditions, or internal 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 timeout event. The ports listed in the dispatch condition must be consistent with the ports listed in the core AADL property associations 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.
- (19) In other categories of components than threads and subprograms, external conditions specify transition conditions out of *execution* states, triggered on reception of events, or event data, on event or event data ports. For instance, this type of condition can be used to represent mode transitions in process components.

- (20) The guard of a transition may rely on an internal condition in order to abstract the reaction of a component on the occurrence of internal events (or internal event data). The objective is to enable interactions among different annexes of the same component. The abstract behavior based on internal conditions may be refined in another Behavior_Specification annex subclause by describing under which circumstances such internal events (or internal event data) can be sent. The internal condition is specified by a list of internal events (or internal event data): the transition is fired if at least one of the internal events (or internal event data) was used. In this case, all the listed internal events (or internal event data) are reset to their initial configuration (i.e., event is considered as not present).
- (21) 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.
- (22) 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, branches, loops and time consuming activities. An action is related to the transition and not to the states: if a transition is taken, actions are performed and then the state specified as the destination of the transition becomes the new current state.

Semantics

- (23) A Behavior_Specification 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.
- (24) A Behavior_Specification 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_Specification 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.
- (25) The core AADL standard defines runtime states for threads using timed automata [AS5506C, 5.4.1]. These states include an *initial* state when the thread not yet begun (thread halted), a *complete* state (awaiting dispatch) and a *final* state when the thread stopped (thread halted). The behavior annex specification for a thread must be consistent with the core AADL semantics.
- (26) 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.
- (27) In a behavior specification, the *initial* state can be a *complete* state (i.e., an *initial complete* state). Such a state is an implicit superposition of two states - an *initial* state and a *complete* state - connected by an implicit transition. This implicit transition – from the implicit *initial* state towards the *implicit complete* state - is triggered by the initialize action defined in the core standard. No condition can be associated to this implicit transition. No other action than the initialization action defined in the core standard (i.e., call to the *Initialize_Entrypoint* as defined by a property in the core language) can be associated to the implicit transition. Note that entering (resp. exiting) an *initial complete* state stands for entering (resp. exiting) the implicit *complete* state. This means that no transition can reach the implicit *initial* state.

- (28) A dispatch trigger might cause a transition out of a *complete* state and to one of the states defined in the behavior annex (either an *execution* state or a *complete* state). A dispatch trigger can be the arrival of input on ports, a subprogram call initiated by another thread, or a timed event in case of a *periodic*, *hybrid* or *timed* Dispatch_Protocol. 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 when a dispatch condition of a transition leaving that state is met.
- (29) 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 or 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.
- (30) The dispatch conditions are evaluated to determine whether a dispatch occurs. If there are multiple outgoing transitions each dispatch condition 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.
- (31) When the Dispatch_Protocol property is *timed* or *hybrid*, the value of the timeout dispatch condition is given by the Period property of the component.
- (32) 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.
- (33) Transitions leaving *execution* states may have transition condition with Boolean expressions, timeout, or *otherwise*. An *otherwise* execute condition becomes true when all the other execute conditions associated with transitions from the same state are false. A timeout execute condition becomes true when the time since last dispatch exceeds the Period property of threads with *hybrid* or *timed* dispatch protocols. When no *execute* condition is specified (i.e., is blank) the condition is assumed to be true. When multiple *execute* conditions of transitions leaving the same execution state are true, the highest priority condition is taken (if any); otherwise which transition is taken is non-deterministic

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
  [ := value_constant ]
  [ { { data_classifier_property_association }+ } ] ;

property_association ::= <refer to [AS5506C 11.3]>

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
| external_condition
| internal_condition

execute_condition ::=
  [ logical_value_expression | behavior_action_block_timeout_catch | otherwise ]

timeout_catch ::=
  timeout

declarator ::=
  identifier { array_size }*

array_size ::=
  [ integer_value_constant ]

external_condition ::=
  on trigger_logical_expression

trigger_logical_expression ::=
  event_trigger { logical_operator event_trigger }*

event_trigger ::=
  in_event_port_component_reference
| in_event_data_port_component_reference
| ( trigger_logical_expression )

port_component_reference ::=
  [ subcomponent_name . ] port_identifier

internal_condition ::= on internal internal_port_name { or internal_port_name }*

```

Naming Rules

- (N1) The variable, state and transition identifiers must be unique within a Behavior_Specification annex subclause. In addition, they must not conflict with mode identifiers, feature identifiers, or data subcomponent identifiers of the component for which the Behavior_Specification annex subclause is defined. An exception to this rule applies for identifiers of 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 not define any *complete* states.
- (L2) 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.