# Annex Document Y - Assertion

Normative
v0.9
April 26, 2016

0



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

# Introduction

(1) This document is an annex standard of SAE Standard AS5506B, Architecture Analysis and Design Language (AADL) to define an annex sublanguage of AADL defining declarative, temporal logic, assertions.

(2) This *assertion* language was developed as part of the Behavior Language for Embedded Systems with Software (BLESS), but has been standardized separately from BLESS so that it can be used independently.

(3) Assertions can be attached to ports, as properties, to specify what is guaranteed to be true of events or data emitted from an out port, or assumed about events or data received by an in port. Assertions can also be used to define component invariants, much like loop invariants. Thus a component's behavior can be formally, declaratively, specified by its port assertion properties and its invariant property.

(4) Assertions may have labels. Labeled assertions may be referenced by other assertions, possible with parameters. Assertion labels may be replaced in other assertions with their bodies, having actual parameters substituted for formal parameters, if any.

(5) Assertions are basically first-order predicates, that have been augmented with simple temporal operators that determine when predicates are evaluated.

# Contents

# Chapter Y.1

# Scope

# Chapter Y.2

# Assertion

(1) Assertion properties may be attached to AADL component features, behavior states, interlaced through actions, or express invariants, and have three forms: predicates, functions, and enumerations.

(2) *Assertion annex libraries* hold labelled Assertions in AADL packages.

(3) *Assertion-predicates* declare truth.

(4) *Assertion-functions* declare value. Assertion-functions specify meaning for data ports or other things with value, or used with other Assertion-functions or Assertions.

(5) Meaning for enumeration-typed ports and variables use *Assertion-enumerations* –a kind of Assertion-function with special grammar associating enumeration identifiers with predicates.

## Annex Y.2.1 Assertion Annex Library

(1) AADL packages may have annex libraries, not attached to any particular component.[1] An annex library is distinguished by the reserved word **annex**, followed by the identifier of the annex, and user-defined text between {** and **}, terminated with a semicolon.

(2) An Assertion annex library contains at least one Assertion.

```
assertion_annex_library ::= annex Assertion {** { assertion }+ **} ;
```

*Example*

AADL source code for an Assertion annex library used in the definition of behavior of a pulse oximeter:

```
annex Assertion
{** --annex library holding BLESS Assertions
```

[1] AS5506B §4.8 Annex Subclauses and Annex Libraries

```
  <<SPO2_LOWER_LIMIT_ALARM: :SensorConnected and not MotionArtifact and
    (SpO2 < SpO2LowerLimit)>>
  <<HEART_RATE_LOWER_LIMIT_ALARM: :SensorConnected and not MotionArtifact and
    (HeartRate < HeartRateLowerLimit)>>
  <<HEART_RATE_UPPER_LIMIT_ALARM: :SensorConnected and not MotionArtifact and
    (HeartRate > HeartRateUpperLimit)>>
  <<SPO2_AVERAGE: :=
     --the sum of good SpO2 measurements
      (sum i:integer in -SpO2MovingAvgWindowSamples..-1 of
         (SensorConnected^(i) and not MotionArtifact^(i)??SpO2^(i):0))
      /  --divided by the number of good SpO2 measurements
      (numberof i:integer in -SpO2MovingAvgWindowSamples..-1
       that (SensorConnected^(i) and not MotionArtifact^(i)))>>
  <<SUPPL_O2_ALARM: :SupplOxyAlarmEnabled^0 and
    (SPO2_AVERAGE())^0 < (SpO2LowerLimit^0+SpO2LevelAdj^0)>>
  <<RAPID_DECLINE_ALARM: :AdultRapidDeclineAlarmEnabled  and
      (exists j:integer in 1 .. NUM_WINDOW_SAMPLES()
       that (SpO2 <= (SpO2^(-j) - MaxSpO2Decline)))>>
  <<MOTION_ARTIFACT_ALARM: :all j:integer
       in 0 ..PulseOx_Properties::Motion_Artifact_Sample_Limit
       are (MotionArtifact^(-j) or not SensorConnected^(-j))>>
  <<SPO2_TREND: : all s:integer in 1 ..num_samples
        are SpO2Trend[s]=(MotionArtifact^(-s) or
          not SensorConnected^(-s)??0:SpO2^(-s))>>
  <<HR_TREND: : all s:integer in 1 ..num_samples are HeartRateTrend[s]=
     (MotionArtifact^(-s) or not SensorConnected^(-s)??0:HeartRate^(-s))>>
  <<AXIOM_CR: :(num_samples-2)<(num_samples-1)>>
**};
```

## Annex Y.2.2 Assertion

(1) In Behavior Language for Embedded Systems with Software (BLESS), an *Assertion* is a temporal logic formula enclosed between << and >>.

```
assertion ::=
  << ( assertion_predicate
     | assertion_function
     | assertion_enumeration
     | assertion_enumeration_invocation ) >>
```

### Annex Y.2.2.1 Formal Assertion Parameter

(1) Assertions may have formal parameters.

```
formal_assertion_parameter ::= parameter_identifier [ ~ type_name ]
formal_assertion_parameter_list ::= formal_assertion_parameter { (,) formal_assertion_p
```

Types for assertion parameters may be data component names, or the reserved word for one of the built-in BLESS types. Types and type checking is defined in .

```
type_name ::=
  { package_identifier :: }* data_component_identifier
    [ . implementation_identifier ]
  | natural | integer | rational | real
  | complex | time | string
```

## Annex Y.2.2.2 Assertion-Predicate

(1) Most Assertions will be predicates and may have a label by which other Assertions can refer to it. An *assertion-predicate* may have formal parameters. If so an assertion-predicate's meaning is textual substitution of actual parameter for formal parameters throughout the body of the Assertion.[2]

```
assertion_predicate ::=
   [ label_identifier : [ formal_assertion_parameter_list ] : ] predicate
```

(2) If an Assertion has no parameters, occurrences of its invocation may be replaced by the text of its predicate. If a Assertion has parameters, its label and actual parameters, may be replaced by its predicate with formal parameters replaced by actual parameters.

(3) Any entity may have its `BLESS::Assertion` property associated with the label of an Assertion in a Assertion annex library.

(4) Semantics for use of Assertion-predicates, substitution of actual parameters for formal parameters, is defined in Y.2.3.5, Predicate Invocation.

*Example*

AADL source code for Assertions used in the definition of behavior of a cardiac pacemaker:

```
<<LRL:x:   --Lower Rate Limit
    -- there has been a V-pace or a non-refractory V-sense
    exists t:BLESS_Types::Time
      -- within the previous LRL interval
      in (x-max_cci)..x   --MaxCCI is the maximum cardiac cycle interval
      -- in which a heartbeat was sensed, or caused by pacing
      that (vs or vp)@t  >>
<<LAST_A_WAS_AS:x: exists t:BLESS_Types::Time in x-max_cci..x that
  (as@t and  --A-sense at time t
    not (exists t2:BLESS_Types::Time in t,,x that  --no as or ap since
    (as@t2 or ap@t2))) >>
<<ATR_DURATION:d dur_met:  --wait to be sure a-tachy continues
  ATR_DETECT(d) and  --detection met at time d
  (dur > (numberof t:BLESS_Types::Time in d..dur_met that (vs@t or sp@t)))
  and  (all t2:BLESS_Types::Time in d..dur_met are not ATR_END(t2)) >>
```

[2] If an Assumption has a label, but no parameters, leave a space between to colons so the lexical analyzer emits two colon tokens, not one double-colon token.

### Annex Y.2.2.3 Assertion-Function

(1) An *Assertion-function* abstracts a value, usually numeric. Labeled Assertion-functions may be used in Assertion-expressions.

```
assertion_function ::=
    [ label_identifier : [ formal_assertion_parameter_list ] ]
    :=  ( assertion_expression | conditional_assertion_function )
```

(2) Semantics for use of Assertion-functions, substitution of actual parameters for formal parameters, is defined in Y.2.4.6, Assertion Function Invocation.

*Example*

An Assertion-function defining a moving average, neglecting bad measurements:

```
<<SPO2_AVERAGE: :=
   --the sum of good SpO2 measurements
    (sum i:integer in -SpO2MovingAvgWindowSamples..-1 of
       (SensorConnected^(i) and not MotionArtifact^(i)??SpO2^(i):0))
    /  --divided by the number of good SpO2 measurements
    (numberof i:integer in -SpO2MovingAvgWindowSamples..-1
      that (SensorConnected^(i) and not MotionArtifact^(i)))>>
```

An Assertion-function that determines the maximum cardiac cycle interval during atrial tachycardia response fall back:

```
<<FallBack_MaxCCI:dur_met x:= (x-dur_met)*((lrl-url)/fb_time)>>
```

### Annex Y.2.2.4 Assertion-Enumeration

(1) An *Assertion-enumeration* associates an Assertion with elements (identifiers) of enumeration types. Assertion-enumerations are usually used as a data port property having enumeration type to define what is true about the system for different elements.

(2) An Assertion-enumeration has one parameter for the enumeration value sent or received by an event data port

```
assertion_enumeration ::=
   asserion_enumeration_label_identifier :  parameter_identifier +=>
    enumeration_pair { , enumeration_pair }*

enumeration_pair ::= enumeration_literal_identifier -> predicate
```

(3) Semantics for use of Assertion-enumerations, selection of enumeration pair matching given enumeration value, is defined in Y.2.4.7, Assertion Enumeration Invocation.

*Example*

```
<<ALARM_TYPE: x +=> --has enumeration value of first element
          --when predicate in 2nd element is true
```

```
  Pump_Overheated->PUMP_OVERHEATED,
  Defective_Battery->DEFECTIVE_BATTERY,
  Low_Battery->LOW_BATTERY,
  POST_Failure->POST_FAIL,
  RAM_Failure->RAM_FAIL,
  ROM_failure->ROM_FAIL,
  CPU_Failure->CPU_FAIL,
  Thread_Monitor_Failure->THREAD_MONITOR_FAIL,
  Air_In_Line->AIR_IN_LINE,
  Upstream_Occlusion->UPSTREAM_OCCLUSION,
  Downstream_Occlusion->DOWNSTREAM_OCCLUSION,
  Empty_Reservoir->EMPTY_RESERVOIR,
  Basal_Overinfusion->BASAL_OVERINFUSION,
  Bolus_Overinfusion->BOLUS_OVERINFUSION,
  Square_Bolus_Overinfusion->SQUARE_OVERINFUSION,
  No_Alarm->NO_ALARM >>
```

## Annex Y.2.3 Predicate

(1) A *predicate* is a boolean valued function, when evaluated returns *true* or *false*. A Assertion claims its predicate is *true*. The meaning of the logical operators within a predicate have customary meanings. Universal quantification is defined in Y.2.3.8, and existential quantification is defined in D Y.2.3.9.

```
predicate ::=
   universal_quantification | existential_quantification |
   subpredicate
      [ { and subpredicate }+
      | { or subpredicate }+
      | { xor subpredicate }+
      | implies subpredicate
      | iff subpredicate
      | -> subpredicate ]
```

*Semantics*

(S1) Where i is an interval, and A,B are predicate atoms:

$\mathfrak{M}_{i}[\![A \textbf{ and } B]\!] \equiv \mathfrak{M}_{i}[\![A]\!] \wedge \mathfrak{M}_{i}[\![B]\!]$ *(the meaning of* **and** *is conjunction)*
$\mathfrak{M}_{i}[\![A \textbf{ or } B]\!] \equiv \mathfrak{M}_{i}[\![A]\!] \vee \mathfrak{M}_{i}[\![B]\!]$ *(the meaning of* **or** *is disjunction)*
$\mathfrak{M}_{i}[\![A \textbf{ xor } B]\!] \equiv \mathfrak{M}_{i}[\![A]\!] \oplus \mathfrak{M}_{i}[\![B]\!]$ *(the meaning of* **xor** *is exclusive-disjunction)*
$\mathfrak{M}_{i}[\![A \textbf{ implies } B]\!] \equiv \mathfrak{M}_{i}[\![A]\!] \rightarrow \mathfrak{M}_{i}[\![B]\!]$ *(the meaning of* **implies** *is implication)*
$\mathfrak{M}_{i}[\![A \textbf{ iff } B]\!] \equiv \mathfrak{M}_{i}[\![A]\!] \leftrightarrow \mathfrak{M}_{i}[\![B]\!]$ *(the meaning of* **iff** *is if-and-only-if)*
$\mathfrak{M}_{i}[\![A \textbf{ -> } B]\!] \equiv \mathfrak{M}_{i}[\![A]\!] \rightarrow \mathfrak{M}_{i}[\![B]\!]$ *(the meaning of* **->** *is implication)*

*Example*

```
<<(goodSamp[ub mod PulseOx_Properties::Max_Window_Samples] iff
```

```
(SensorConnected^0 and not MotionArtifact^0)) and GS()>>
```

### Annex Y.2.3.1 Subpredicate

(1) The meaning of `true`, `false`, and `not` within a predicate have customary meanings. Both parenthesized predicate and name may be followed by a time expression. Being able to express when a predicate will be true makes this a temporal logic able to express useful properties of embedded systems. Predicate invocation is defined in D Y.2.3.5.

(2) The reserved word `def` defines a “logic variable” that represents an unknown, or changing value.

```
subpredicate ::=
   [ not ]
   ( true |  false |  stop
   | predicate_relation
   | timed_predicate
   | event_expression
   | def logic_variable_identifier )
```

*Semantics*

(S2) Where i is an interval, and A is the rest of a subpredicate:

$\mathfrak{M}_{i}[\![\texttt{not}\ A]\!] \equiv \neg\mathfrak{M}_{i}[\![A]\!]$ *(the meaning of **not** is negation)*
$\mathfrak{M}[\![\texttt{def}\ D]\!] \equiv \exists D$ *(the meaning of **def** is definition)*
$\mathfrak{M}[\![\texttt{stop}]\!] \equiv$ **stop?**
*(the meaning of **stop** is arrival of event at pre-declared* stop port *implicit for all AADL components)*

### Annex Y.2.3.2 Timed Predicate

(1) In a *timed predicate*, the time when the predicate holds may be specified. The `'` means the predicate will be true one clock cycle (or thread period) hence; the `@` means the predicate is true when the subexpression, in seconds, is the current time; and the `^` means the predicate is true an integer number of clock ticks from `now`. Grammatically, time expression (Y.2.3.3) and period-shift (D Y.2.3.4) are time-free (e.g. no `'` `@` or `^` within). Grammar and meaning of a name is defined in **??** Name.

```
timed_predicate ::=
  ( name | parenthesized_predicate | predicate_invocation )
  [ ' | @ time_expression | ^ integer_expression ]
```

*Legality Rules*

(L1) When using `@`, the subexpression must have a time type such as, `Timing_Properties::Time`.

(L2) When using `^`, the value must have integer type.

*Semantics*

(S3) Where $P$ is a name or a parenthesized predicate, $t$ is a time, $d$ is the duration of a thread's period, and $k$ is a period-shift:

$\mathfrak{M}[\![\texttt{P@t}]\!] \equiv \mathfrak{M}_t[\![\texttt{P}]\!]$ *(the meaning of P@t is the meaning of P at time t)*
$\mathfrak{M}_t[\![\texttt{P\^{}k}]\!] \equiv \mathfrak{M}_{t+dk}[\![\texttt{P}]\!]$
*(the meaning of P^k at time t, is the meaning of P, k period durations hence, or earlier if $k < 0$)*
$\mathfrak{M}_t[\![\texttt{P'}]\!] \equiv \mathfrak{M}_t[\![\texttt{P\^{}1}]\!] \equiv \mathfrak{M}_{t+d}[\![\texttt{P}]\!]$ *(the meaning of P' at time t, is the meaning of P a period duration hence)*

*Example*

```
<<VS:x: --ventricular sense
  sv@x   --sensing ventricle enabled
  and v@x  --v-signal
  and not tnv@x  --not noisy
  and VRP_EXPIRED(x) >>  --not ventricular refractory period

  <<HR_TREND: : all s:integer in 1..num_samples
         are HeartRateTrend[s]=(MotionArtifact^(-s)
           or not SensorConnected^(-s)??0:HeartRate^(-s))>>
```

### Annex Y.2.3.3 Time-Expression

(1) Both timed predicate (Y.2.3.2 Timed Predicate) and timed expression (Y.2.4.1 Timed Expression) require a *time-expression* when using @ to define when a predicate holds. A time-expression must have type **time**, and must not use @.

```
time_expression ::=
  time_subexpression
  | time_subexpression - time_subexpression
  | time_subexpression / time_subexpression
  | time_subexpression { + time_subexpression
  | time_subexpression { * time_subexpression }+

time_subexpression ::= [ - ]
  ( time_assertion_value
  | ( time_expression )
  | assertion_function_invocation )
```

*Legality Rule*

(L3) Every `time_expression` must have `time` type.

*Semantics*

(S4) Where $e$ and $f$ are time values (real),

$\mathfrak{M}_i[\![\texttt{e+f}]\!] \equiv \mathfrak{M}_i[\![e]\!] + \mathfrak{M}_i[\![f]\!]$ *(the meaning of + is addition)*
$\mathfrak{M}_i[\![\texttt{e*f}]\!] \equiv \mathfrak{M}_i[\![e]\!] \times \mathfrak{M}_i[\![f]\!]$ *(the meaning of * is multiplication)*
$\mathfrak{M}_i[\![\texttt{e-f}]\!] \equiv \mathfrak{M}_i[\![e]\!] - \mathfrak{M}_i[\![f]\!]$ *(the meaning of - is subtraction)*

$\mathfrak{M}_{\mathfrak{i}}[\![\mathtt{e/f}]\!] \equiv \mathfrak{M}_{\mathfrak{i}}[\![e]\!] \div \mathfrak{M}_{\mathfrak{i}}[\![f]\!]$ *(the meaning of / is division)*
$\mathfrak{M}_{\mathfrak{i}}[\![\mathtt{(e)}]\!] \equiv \mathfrak{M}_{\mathfrak{i}}[\![e]\!]$ *(the meaning of parentheses is its contents)*
$\mathfrak{M}_{\mathfrak{i}}[\![\mathtt{-e}]\!] \equiv 0.0 - \mathfrak{M}_{\mathfrak{i}}[\![e]\!]$ *(the meaning of unary minus is complement)*

*Example*

```
<<PACE_ON_MaxCCI:x:     --no intrinsic activity, pace at LRL
  (vp or vs)@(x-max_cci)
      and --and not since
      not (exists t:BLESS_Types::Time
        in x-max_cci,,x
        --with a non-refractory ventricular sense or pace
        that (vs or vp)@t) >>
```

## Annex Y.2.3.4 Period-Shift

(1) Both timed predicate (Y.2.3.2) and timed expression (Y.2.4.1) require a *period-shift* when using ^ to shift its time frame by number of thread periods (a.k.a. clock cycles).

```
integer_expression ::=
  [ - ]
  ( integer_assertion_value
  | ( integer_expression - integer_expression )
  | ( integer_expression / integer_expression )
  | ( integer_expression { + integer_expression }+ )
  | ( integer_expression { * integer_expression }+ ) )
```

*Legality Rule*

(L4) Every `period_shift` must have `integer` type.

*Semantics*

(S5) Where $e$ and $f$ are integers,

$\mathfrak{M}_{\mathfrak{i}}[\![\mathtt{(e+f)}]\!] \equiv \mathfrak{M}_{\mathfrak{i}}[\![e]\!] + \mathfrak{M}_{\mathfrak{i}}[\![f]\!]$ *(the meaning of + is addition)*
$\mathfrak{M}_{\mathfrak{i}}[\![\mathtt{(e*f)}]\!] \equiv \mathfrak{M}_{\mathfrak{i}}[\![e]\!] \times \mathfrak{M}_{\mathfrak{i}}[\![f]\!]$ *(the meaning of * is multiplication)*
$\mathfrak{M}_{\mathfrak{i}}[\![\mathtt{(e-f)}]\!] \equiv \mathfrak{M}_{\mathfrak{i}}[\![e]\!] - \mathfrak{M}_{\mathfrak{i}}[\![f]\!]$ *(the meaning of - is subtraction)*
$\mathfrak{M}_{\mathfrak{i}}[\![\mathtt{(e/f)}]\!] \equiv \mathfrak{M}_{\mathfrak{i}}[\![e]\!] \,/\, \mathfrak{M}_{\mathfrak{i}}[\![f]\!]$ *(the meaning of / is division, neglecting remainder)*
$\mathfrak{M}_{\mathfrak{i}}[\![\mathtt{-e}]\!] \equiv 0 - \mathfrak{M}_{\mathfrak{i}}[\![e]\!]$ *(the meaning of unary minus is complement)*

*Example*

Examples of period shift from a pulse oximeter smart alarm:

```
<<GOOD: :goodCount=(numberof k:integer in lb..ub-1
  that (SensorConnected^(k-ub) and not MotionArtifact^(k-ub)))>>
<<CTR: :(all k:integer in lb..ub-1
    are spo2_hist[k mod PulseOx_Properties::Max_Window_Samples] = C(k-(ub-1)))
  and (totalSpO2=(sum k:integer in lb..ub-1 of C(k-(ub-1))))
  and (goodCount=(numberof k:integer in lb..ub-1
```

```
  that (SensorConnected^(k-(ub-1)) and not MotionArtifact^(k-(ub-1)))))
and (all k:integer in lb..ub-1
  are goodSamp[k mod PulseOx_Properties::Max_Window_Samples] iff
     (SensorConnected^(k-(ub-1)) and not MotionArtifact^(k-(ub-1))))>>;
```

## Annex Y.2.3.5 Predicate Invocation

(1) Predicate invocation allows labeled Assertions to be used by other Assertions.

(2) Predicates of the form `<<B:f:P>>` may be invoked as `B(a)`, where `B` is the label, `f` are formal parameters, `P` is a predicate, and `a` are actual parameters. Predicate invocations with single parameter may omit the formal parameter identifier.

```
predicate_invocation ::= assertion_identifier
    ( [ assertion_expression | actual_assertion_parameter_list ] )

actual_assertion_parameter_list ::=
  actual_assertion_parameter { , actual_assertion_parameter }*

actual_assertion_parameter ::=
  formal_parameter_identifier : actual_parameter_assertion_expression
```

*Semantics*

(S6) Where $B$ is a Assertion label, $f_1 f_2 \ldots f_n$ are formal parameters, and $P$ is a predicate that uses $f_1 f_2 \ldots f_n$, and

$\ll B : f_1 f_2 \ldots f_n : P \gg$ *(there is Assertion B with predicate P & formal parameters f)*

then the meaning of predicate invocation is

$\mathfrak{M}_{\mathfrak{i}}[\![\texttt{B(f1:a1, f2:a2, ... fn:an)}]\!] \equiv \mathfrak{M}_{\mathfrak{i}}[\![B \mid {}^{f_1}_{a_1} \mid {}^{f_2}_{a_2} \cdots \mid {}^{f_n}_{a_n}]\!]$
*(the meaning of a predicate invocation is the meaning of the predicate of the Assertion with the same label having actual parameters substituted for formal parameters)*

*Naming Rule*

(N1) The identifier of a predicate invocation must be the label of a visible or imported Assertion.

*Example*

Examples of predicate invocation from a cardiac pacemaker:

```
<<VP(now) and URL(now)>>
<<ATR_DURATION(d:detect_time, dur_met:now)>>
```

## Annex Y.2.3.6 Predicate Relations

(1) *Predicate relations* have conventional meanings. The `in` operators tests membership of a range.

```
predicate_relation ::=
    assertion_subexpression relation_symbol  assertion_subexpression
  | assertion_subexpression in assertion_range
  | shared_integer_name +=  assertion_subexpression

relation_symbol ::= = | < | > | <= | >= | != | <>
```

(2) The *range* is defined with ordinary subexpressions (??). Ranges may be open or closed on either or both ends.

```
assertion_range ::=
  assertion_subexpression range_symbol assertion_subexpression

range_symbol ::= .. | ,. | ., | ,,
```

*Semantics*

(S7) Where c, d, l, and u are predicate expressions,

$\mathfrak{M}_i[\![c{=}d]\!] \equiv \mathfrak{M}_i[\![c]\!] = \mathfrak{M}_i[\![d]\!]$ *(the meaning of = is equality)*
$\mathfrak{M}_i[\![c{<>}d]\!] \equiv \mathfrak{M}_i[\![c\,!{=}d]\!] \equiv \mathfrak{M}_i[\![c]\!] \neq \mathfrak{M}_i[\![d]\!]$ *(the meaning of <> and != is inequality)*[3]
$\mathfrak{M}_i[\![c{<}d]\!] \equiv \mathfrak{M}_i[\![c]\!] < \mathfrak{M}_i[\![d]\!]$ *(the meaning of < is less than)*
$\mathfrak{M}_i[\![c{>}d]\!] \equiv \mathfrak{M}_i[\![c]\!] > \mathfrak{M}_i[\![d]\!]$ *(the meaning of > is greater than)*
$\mathfrak{M}_i[\![c{<=}d]\!] \equiv \mathfrak{M}_i[\![c]\!] \leq \mathfrak{M}_i[\![d]\!]$ *(the meaning of <= is at most)*
$\mathfrak{M}_i[\![c{>=}d]\!] \equiv \mathfrak{M}_i[\![c]\!] \geq \mathfrak{M}_i[\![d]\!]$ *(the meaning of >= is at least)*
$\mathfrak{M}_i[\![c \textbf{ in } l..u]\!] \equiv \mathfrak{M}_i[\![c]\!] \geq \mathfrak{M}_i[\![l]\!] \wedge \mathfrak{M}_i[\![c]\!] \leq \mathfrak{M}_i[\![u]\!]$ *(the meaning of .. is closed interval)*
$\mathfrak{M}_i[\![c \textbf{ in } l,.u]\!] \equiv \mathfrak{M}_i[\![c]\!] > \mathfrak{M}_i[\![l]\!] \wedge \mathfrak{M}_i[\![c]\!] \leq \mathfrak{M}_i[\![u]\!]$ *(the meaning of ,. is open-left interval)*
$\mathfrak{M}_i[\![c \textbf{ in } l.,u]\!] \equiv \mathfrak{M}_i[\![c]\!] \geq \mathfrak{M}_i[\![l]\!] \wedge \mathfrak{M}_i[\![c]\!] < \mathfrak{M}_i[\![u]\!]$ *(the meaning of ., is open-right interval)*
$\mathfrak{M}_i[\![c \textbf{ in } l,,u]\!] \equiv \mathfrak{M}_i[\![c]\!] > \mathfrak{M}_i[\![l]\!] \wedge \mathfrak{M}_i[\![c]\!] > \mathfrak{M}_i[\![u]\!]$ *(the meaning of ,, is open interval)*

(S8) Where v is an identifier of a shared integer variable, and e is an integer-valued expression,

$\mathfrak{M}_i[\![v \mathrel{+}= e]\!] \equiv \mathfrak{M}_{end(i)}[\![v]\!] = \mathfrak{M}_{start(i)}[\![v]\!] + \mathfrak{M}_{start(i)}[\![e]\!]$ *(the meaning of += is add to total [4])*

## Annex Y.2.3.7 Parenthesized Predicate

(1) Parentheses disambiguate precedence.

```
parenthesized_predicate ::= ( predicate )
```

*Semantics*

(S9) Where $P$ is a predicate,

[3] **Reconciliation:** inequality

[4] The definition of a single **+=** is straight forward: at the end of the interval, the target will be the target value at the beginning of the interval, plus an expression also valued at the beginning of the interval. Defining concurrent **+=** to the same target, in the same interval, is just like solitary **+=**, using the sum of all concurrent expressions. Concurrent **+=** predicate defines concurrent fetch-add action. Fetch-add is used to access shared data structures without locks, allowing unlimited speed-up. See U.S Pat. No. 5,867,649 DANCE-Multitude Concurrent Computation

$\mathfrak{M}_{\text{i}}[\![(P)]\!] \equiv \mathfrak{M}_{\text{i}}[\![P]\!]$ *(the meaning of parenthesis is its contents)*

## Annex Y.2.3.8 Universal Quantification

(1) Universal quantification claims its predicate is true for all the members of a particular set. Logic variables must have types. Bounding the domain of quantification to a range, or when some predicate is true, defines the set of values that variables may take.[5] Quantified variables of type time are particularly useful for declaratively expression cyber-physical systems (CPS). A particular combination of events either did or did not occur in a particular interval of time, or what is true about system state during a particular interval of time.

```
universal_quantification ::=
   all logic_variables logic_variable_domain
   are predicate

logic_variables ::= logic_variable_identifier { , logic_variable_identifier }* : type

logic_variable_domain ::= in
  ( assertion_expression range_symbol assertion_expression
  | predicate )
```

*Semantics*

(S10) Where $v$ is a logic variable, $T$ is an Assertion-type, $R$ is a range, and $P(v)$ is a predicate that uses $v$,

$\mathfrak{M}_{\text{i}}[\![$`all v:T in R are P(v)`$]\!] \equiv \forall\, v \in \mathfrak{M}_{\text{i}}[\![R]\!] \subseteq \mathfrak{M}_{\text{i}}[\![T]\!] \mid \mathfrak{M}_{\text{i}}[\![P(v)]\!]$
*(for all v in R, a subset of T, P(v) is true)*

*Example*

```
<<MOTION_ARTIFACT_ALARM: :all j:integer
      in 0..PulseOx_Properties::Motion_Artifact_Sample_Limit
      are (MotionArtifact^(-j) or not SensorConnected^(-j))>>
```

## Annex Y.2.3.9 Existential Quantification

(1) Existential quantification claims its predicate is true for at least one member of a particular set.

```
existential_quantification ::=
   exists logic_variables logic_variable_domain
   that predicate
```

*Semantics*

(S11) Where $v$ is a logic variable, $T$ is as Assertion-type, $R$ is a range, and $P(v)$ is a predicate that uses $v$,

[5] Bounding quantification is highly recommended.

$\mathfrak{M}_{\mathrm{i}}[\![$**exists** v:T **in** R **that** P(v)$]\!] \equiv \exists\, v \in \mathfrak{M}_{\mathrm{i}}[\![R]\!] \subseteq \mathfrak{M}_{\mathrm{i}}[\![T]\!] \mid \mathfrak{M}_{\mathrm{i}}[\![P(v)]\!]$
*(there exists v in R, a subset of T, for which P(v) is true)*

*Example*

```
<<RAPID_DECLINE_ALARM: :AdultRapidDeclineAlarmEnabled  and
     (exists j:integer in 1..NUM_WINDOW_SAMPLES()
       that (SpO2 <= (SpO2^(-j) - MaxSpO2Decline)))>>
```

### Annex Y.2.3.10 Event

(1) An *event* occurs when either a port or variable has a (non-null) value, or the state machine is in a particular state (see **??** Clock).

```
event ::= < port_variable_or_state_identifier >

event_expression ::= [not] event
  | event_subexpression (and event_subexpression)+
  | event_subexpression (or event_subexpression)+
  | event - event

event_subexpression ::= [ always | never ] ( event_expression ) | event
```

*Semantics*

(S12) Where p is a port identifier <p> $\equiv \hat{p} \equiv \mathfrak{M}_{now}[\![p \neq \bot]\!]$.
Where v is a variable identifier <v> $\equiv \hat{v} \equiv \mathfrak{M}_{now}[\![v \neq \bot]\!]$.
Where s is a state identifier <s> $\equiv \hat{s} \equiv \mathfrak{M}_{now}[\![State(s)]\!]$ where $State(s)$ means the state machine is currently in state $s$.

(S13) Where <x> and <y> are events, <x> - <y> $\equiv \hat{x} \mathbin{\hat{-}} \hat{y}$.

(S14) Where ee is an event expression, **never**(ee)≡ee= $\hat{0}$, and **always**(ee)≡ee= $\hat{1}_{SVP}$.

(S15) Logical operators `not`, `and`, `or` are complement, conjunction, and disjunction, respectively. Parentheses group.

## Annex Y.2.4 Assertion-Expression

(1) Other useful quantifiers add, multiply, or count the elements of sets. There is no operator precedence so parentheses must be used to avoid ambiguity. Numeric operators have their usual meanings.

(2) Assertion-expressions differ from expression usually found in programming languages which are intended to be evaluated during execution. Rather, assertion expressions define values derived from over values, usually numeric. Such predicate expressions usually appear within predicates that contain relations between values. Predicate expressions may also used within Assertion-functions (Y.2.2.3) to define Assertions that return values.

(3) Numeric quantifiers sum, product, and number-of have an optional logic variable domain, but include one whenever possible. Bounding quantification prevents oddities that can occur with infinite domains. In mathematics, sums of an infinite number of ever smaller terms are quite common. But for reasoning about program behavior, stick to bounded quantifications.

```
assertion_expression ::=
   sum logic_variables [ logic_variable_domain ]
      of assertion_expression
   | product logic_variables [ logic_variable_domain ]
      of assertion_expression
   | numberof logic_variables [ logic_variable_domain ]
      that subpredicate
   | assertion_subexpression
      [ { + assertion_subexpression }+
      | { * assertion_subexpression }+
      | - assertion_subexpression
      | / assertion_subexpression
      | ** assertion_subexpression
      | mod assertion_subexpression
      | rem assertion_subexpression ]
```

*Semantics*

(S1) Where $v$ is a logic variable, $T$ is a type, $R$ is a range, $P(v)$ is a predicate that uses $v$, $E(v)$ is a predicate expression that uses $v$, and $e, f$ are predicate subexpressions,

$\mathfrak{M}_i[\![$**sum** v:T **in** R **of** E(v)$]\!] \equiv \sum_{v \in R} \mathfrak{M}_i[\![E(v)]\!]$
*(sum the value E(v) for each v in the range R)*
$\mathfrak{M}_i[\![$**product** v:T **in** R **of** E(v)$]\!] \equiv \prod_{v \in R} \mathfrak{M}_i[\![E(v)]\!]$
*(multiply the value E(v) for each v in the range R)*
$\mathfrak{M}_i[\![$**numberof** v:T **in** R **that** P(v)$]\!] \equiv \|\{v \in \mathfrak{M}_i[\![R]\!] \mid \mathfrak{M}_i[\![P(v)]\!]\}\|$
*(cardinality of the set of v in R for which P(v) is true)*
$\mathfrak{M}_i[\![\texttt{e+f}]\!] \equiv \mathfrak{M}_i[\![e]\!] + \mathfrak{M}_i[\![f]\!]$ *(the meaning of + is addition)*
$\mathfrak{M}_i[\![\texttt{e*f}]\!] \equiv \mathfrak{M}_i[\![e]\!] \times \mathfrak{M}_i[\![f]\!]$ *(the meaning of * is multiplication)*
$\mathfrak{M}_i[\![\texttt{e-f}]\!] \equiv \mathfrak{M}_i[\![e]\!] - \mathfrak{M}_i[\![f]\!]$ *(the meaning of - is subtraction)*
$\mathfrak{M}_i[\![\texttt{e/f}]\!] \equiv \mathfrak{M}_i[\![e]\!] \div \mathfrak{M}_i[\![f]\!]$ *(the meaning of / is division)*
$\mathfrak{M}_i[\![\texttt{e**f}]\!] \equiv \mathfrak{M}_i[\![e]\!]^{\mathfrak{M}_i[\![f]\!]}$ *(the meaning of ** is exponentiation)*
$\mathfrak{M}_i[\![\texttt{e }\textbf{mod}\texttt{ f}]\!] \equiv \mathfrak{M}_i[\![e]\!] \mathrm{\ mod\ } \mathfrak{M}_i[\![f]\!]$ *(the meaning of **mod** is modulus)*
$\mathfrak{M}_i[\![\texttt{e }\textbf{rem}\texttt{ f}]\!] \equiv \mathfrak{M}_i[\![e]\!] \mathrm{\ rem\ } \mathfrak{M}_i[\![f]\!]$ *(the meaning of **rem** is remainder)*

*Legality Rule*

(L1) The ranges for sum, product, and numberof predicate expressions must be discrete and finite.

(4) Predicate subexpressions allow optional negation of a timed expression. Negation has the usual meaning.

```
assertion_subexpression ::=
  [ - | abs ] timed_expression
  | assertion_type_conversion
```

```
assertion_type_conversion ::=
  ( natural | integer | rational | real | complex | time )
  parenthesized_assertion_expression
```

*Semantics*

(S2) Where $S$ is a predicate expression,

$\mathfrak{M}_{\mathfrak{i}}[\![-S]\!] \equiv 0 - \mathfrak{M}_{\mathfrak{i}}[\![S]\!]$ *(the meaning of - is negation)*
$\mathfrak{M}_{\mathfrak{i}}[\![\textbf{abs}\ S]\!] \equiv \mathfrak{M}_{\mathfrak{i}}[\![(\textbf{if}\ S>=0\ \textbf{then}\ S\ \textbf{else}\ -S)]\!]$ *(the meaning of **abs** is absolute value)*[6]

*Example*

```
<<SPO2_AVERAGE: :=
   --the sum of good SpO2 measurements
    (sum i:integer in -SpO2MovingAvgWindowSamples..-1 of
       (SensorConnected^(i) and not MotionArtifact^(i)??SpO2^(i):0))
    /  --divided by the number of good SpO2 measurements
    (numberof i:integer in -SpO2MovingAvgWindowSamples..-1
     that (SensorConnected^(i) and not MotionArtifact^(i)))>>
```

## Annex Y.2.4.1 Timed Expression

(1) In a *timed expression*, the time when the expression is evaluated may be specified. The ' means the value of the expression one clock cycle (or thread period) hence; the @ means the value of the expression when the subexpression (to the right of the @), in seconds, is the current time; and the ^ means the value of the expression an integer number of clock ticks from now. Grammatically, time-expression and period-shift are time-free (no ' @ or ^ within).

```
timed_expression ::=
     ( assertion_value
       | parenthesized_assertion_expression
       | predicate_incocation )
     [ '
     | ^ integer_expression
     | @ time_expression ]
```

*Legality Rules*

(L2) When using @, the subexpression must have a time type such as, Timing_Properties::Time.

(L3) When using ^, the value must have integer type.

*Semantics*

(S3) Where $E$ is a value, a parenthesized predicate expression, or a conditional predicate expression, $t$ is a time, $d$ is the duration of a thread's period, and $k$ is an integer:

[6] **Reconciliation:** absolute value

$\mathfrak{M}[\![\texttt{E@t}]\!] \equiv \mathfrak{M}_t[\![\texttt{E}]\!]$ *(the meaning of* E@t *is the meaning of* E *at time* $t$*)*
$\mathfrak{M}_t[\![\texttt{E\^{}k}]\!] \equiv \mathfrak{M}_{t+dk}[\![\texttt{E}]\!]$ *(the meaning of* E^k *at time* $t$*, is the meaning of* E*,* $k$ *period durations hence, or earlier if* $k < 0$*)*
$\mathfrak{M}_t[\![\texttt{E'}]\!] \equiv \mathfrak{M}_t[\![\texttt{E\^{}1}]\!] \equiv \mathfrak{M}_{t+d}[\![\texttt{E}]\!]$ *(the meaning of* E' *at time* $t$*, is the meaning of* E *a period duration hence)*

*Example*

```
<<heart_rate[i]=(MotionArtifact^(1-i) or not SensorConnected^(1-i)
    ??0:HeartRate^(1-i))>>
```

## Annex Y.2.4.2 Parenthesized Assertion Expression

(1) Parentheses around assertion expressions determine operator precedence. Both conditional assertion expressions and record term have inherent parentheses.

```
parenthesized_assertion_expression ::=
    ( assertion_expression )
    | conditional_assertion_expression
    | record_term
```

## Annex Y.2.4.3 Assertion-Value

(1) An *Assertion-value* is atomic, so cannot be further subdivided into simpler expressions. The value of *tops* is the time of previous suspension of the thread which contains it; tops is used commonly in expressions of timeouts. The value of Assertion function invocation is given in Y.2.3.5. Property values according to AS5506B §11 Properties. Port values according to AS5506B §8.3 Ports.

```
assertion_value ::=
  now | tops | timeout
  | value_constant
  | variable_name
  | assertion_function_invocation
  | port_value
```

## Annex Y.2.4.4 Conditional Assertion Expression

(1) A *conditional assertion expression* determines the value of a predicate expression by evaluating a boolean expression or relation, then choosing between alternative expressions, having the first value if true or the second value if false.

```
conditional_assertion_expression ::=
  ( predicate ?? assertion_expression : assertion_expression )
```

*Semantics*

(S4) Where t and f are expressions and B is a boolean-valued expression or relation:

$$\mathfrak{M}_{\mathrm{i}}[\![\texttt{(B??t:f)}]\!] \equiv \begin{array}{l} \mathfrak{M}_{\mathrm{i}}[\![\texttt{B}]\!] \rightarrow \mathfrak{M}_{\mathrm{i}}[\![\texttt{t}]\!] \\ \neg\mathfrak{M}_{\mathrm{i}}[\![\texttt{B}]\!] \rightarrow \mathfrak{M}_{\mathrm{i}}[\![\texttt{f}]\!] \end{array}$$

*(choose first value if true; second value if false)*

*Example*

```
<<(all i:integer in 1 ..num_samples
  are spo2[i]'=(if MotionArtifact^(1-i) or not SensorConnected^(1-i)
        then 0 else SpO2^(1-i)))
      and (num_samples'=PulseOx_Properties::Num_Trending_Samples)>>
```

## Annex Y.2.4.5 Conditional Assertion Function

(1) A *conditional assertion function* is much like a conditional assertion expression (Y.2.4.4), but allows an arbitrary number of choices, each of which is controlled by a predicate. A conditional assertion function is only permitted as a Assertion-function value (Y.2.2.3).

(2) Conditional Assertion-function was added to specify the flow rate of a patient-controlled analgesia (PCA) pump. Rather than a smooth function, the flow rate must be different depending on system state (see example). PUMP_RATE is the BLESS::Assertion property of a port of the thread deciding infusion rate. Each of the parenthesized predicates embodies complex conditions that must be true for each of the possible infusion rates. When a value is output from the port, a proof obligation is generated to ensure that the corresponding property holds.

```
conditional_assertion_function ::=
    condition_value_pair { , condition_value_pair }*

condition_value_pair ::=
    parenthesized_predicate -> assertion_expression
```

*Semantics*

(S5) Where C1, C2, and C2 are predicates and E1, E2, and E3 are Assertion-expressions:

$$\mathfrak{M}_{\mathrm{i}}[\![\texttt{(C1) ->E1, (C2) ->E2, (C3) ->E3}]\!] \equiv \begin{array}{l} \mathfrak{M}_{\mathrm{i}}[\![\texttt{C1}]\!] \rightarrow \mathfrak{M}_{\mathrm{i}}[\![\texttt{E1}]\!] \\ \mathfrak{M}_{\mathrm{i}}[\![\texttt{C2}]\!] \rightarrow \mathfrak{M}_{\mathrm{i}}[\![\texttt{E2}]\!] \\ \mathfrak{M}_{\mathrm{i}}[\![\texttt{C3}]\!] \rightarrow \mathfrak{M}_{\mathrm{i}}[\![\texttt{E3}]\!] \end{array}$$

*(choose the value corresponding to the true condition)*

*Example*

Conditional Assertion-functions should be used sparingly. The pump-rate example below induced conditional Assertion-function's creation to define infusion rate in different conditions.

```
<<PUMP_RATE: :=
  (HALT()) -> 0,                                      --no flow
  (KVO_RATE()) -> PCA_Properties::KVO_Rate,           --KVO rate
```

```
  (PB_RATE()) -> PCA_Properties::Patient_Button_Rate, --maximum infusion
  (CCB_RATE()) -> Square_Bolus_Rate,                   --square bolus rate
  (PRIME_RATE()) -> PCA_Properties::Prime_Rate,        --pump priming
  (BASAL_RATE()) -> Basal_Rate                         --basal rate, from data port
>>
```

## Annex Y.2.4.6 Assertion-Function Invocation

Assertion-functions which are declared in the form `<<C:f:=E>>` and may be invoked like functions as a predicate value `C(a)`, where

- `C` is the label,
- `f` are formal parameters,
- `E` is an Assertion-expression, and
- `a` are actual parameters.

```
assertion_function_invocation ::=
  assertion_function_identifier
    ( [ assertion_expression |
      actual_assertion_parameter { , actual_assertion_parameter }* ] )
actual_assertion_parameter ::=
  formal_identifier : actual_assertion_expression
```

*Semantics*

(S6) Where $C$ is an Assertion-function label, $f_1 f_2 \ldots f_n$ are formal parameters, and $E$ is a predicate expression that uses $f_1 f_2 \ldots f_n$, and

$\ll C : f_1 f_2 \ldots f_n := E \gg$
*(there is Assertion-function C with predicate expression E and formal parameters f)*

(S7) The meaning of Assertion-function invocation is

$\mathfrak{M}_{\mathrm{i}}[\![\texttt{C(a1 a2 ... an)}]\!] \equiv \mathfrak{M}_{\mathrm{i}}[\![E \mid ^{f_1}_{a_1} \mid ^{f_2}_{a_2} \cdots \mid ^{f_n}_{a_n}]\!]$
*(the meaning of an assertion function invocation is the meaning of the expression of the Assertion-function with the same label having actual parameters substituted for formal parameters)*

*Example*

```
<<SUPPL_O2_ALARM: :SupplOxyAlarmEnabled^0 and
  (SPO2_AVERAGE())^0 < (SpO2LowerLimit^0+SpO2LevelAdj^0)>>
```

## Annex Y.2.4.7 Assertion-Enumeration Invocation

Assertion-enumerations which are declared in the form `<<C:x+=>R>>` and may be invoked like functions as a predicate value `C(a)`, where

- `C` is the label of the Assertion-enumeration,
- `a` is an enumeration-element identifier, and
- `R` is a set of enumeration pairs (label->predicate).

```
assertion_enumeration_invocation ::=
  +=> assertion_enumeration_label_identifier
     ( actual_assertion_parameter )
```

*Semantics*

(S8) Where

$C$ is an Assertion-enumeration label,

$L$ is a set of enumeration labels $\{l_1, l_2, \ldots, l_n\}$,

$a$ is the formal parameter, an enumeration label $a \in L$,

$P$ is a set of predicates $\{p_1, p_2, \ldots, p_n\}$, and

$R$ is a set of enumeration pairs, $\{l_1 \rightarrow p_1, l_2 \rightarrow p_2, \ldots, l_n \rightarrow p_n\}$ defining the onto relation[7] between enumeration labels and their meaning, $R(j) = q \equiv j \rightarrow q \in R$

and

`<<C:x+=>R>>` *(there is Assertion-enumeration C with enumeration pairs R and ignored parameter x)*

(S9) The meaning of Assertion-enumeration invocation is

$\mathfrak{M}_{\mathfrak{i}}[\![\texttt{C(a)}]\!] \equiv \mathfrak{M}_{\mathfrak{i}}[\![R(a)]\!]$
*(the meaning of an Assertion-enumeration invocation is the predicate paired with given label $a$)*

*Example*

(1) Enumeration types should be used sparingly. Assertion-enumerations were created to express the meaning of event-data with enumeration type. Ports having enumeration types may only have enumeration literals for out parameters. The following example expressed the meaning of ‘On’ and ‘Off’ in section A.5.1.3 of the isolette example in FAA’s Requirement Engineering Management Handbook:

```
--A.5.1.3 Manage Heat Source Function
<<HEAT_CONTROL:x+=>
  On  -> REQMHS2() or --below desired range
         (REQMHS4() and (heat_control^-1=On)),
  Off -> REQMHS1() or --initialization
         REQMHS3() or --above desired range
         REQMHS5() or --failed
         (REQMHS4() and (heat_control^-1=Off)) >>
```

Used to define the meaning of the value of port `heat_control`:

[7] Every label has exactly one predicate defining its meaning.

```
heat_control : out data port Iso_Variables::on_off
  {BLESS::Assertion => "<<+=>HEAT_CONTROL(x)>>";};
```

When an enumeration value is sent out port in state-machine action:

```
  mhsBelow:  --REQ-MHS-2  temp below desired range
  check_temp -[current_temperature? <= lower_desired_temperature?]-> run
    { <<REQMHS2() and not REQMHS1()>>
    heat_control!(On)  --temp below desired range
    ; <<heat_control=On>>
    heat_previous_period' := On
      <<heat_previous_period' = heat_control>>
    };  --end of mhsBelow
```

During transformation from proof outline to complete proof, port output of 'On' and its precondition

```
<<REQMHS2() and not REQMHS1()>>
    heat_control!(On)  --temp below desired range
```

becomes a verification condition, that what's claimed for 'On' holds

```
<<REQMHS2() and not REQMHS1()>>
->
<<REQMHS2() or (REQMHS4() and (heat_control^-1=On))>>
```

(2) If it's just two labels (off/on) use a simple predicate instead. Save the hassle of putting meaning to enumeration labels for when it's unavoidable:

```
--regulator mode Figure A-4. Regulate Temperature Mode Transition Diagram
<<REGULATOR_MODE:x+=>
  Init   -> INI(),
  NORMAL -> REGULATOR_OK() and RUN(),
  FAILED -> not REGULATOR_OK() and RUN() >>
```

# Chapter Y.3

# Names and Values

## Annex Y.3.1 Value Constant

(1) Value constants are Boolean, numeric or string literals, property constants or property values.[1]

```
value_constant ::=
  true | false | numeric_literal | string_literal
  | property_constant | property_reference
```

(2) Literals follow AS5506B §15 Lexical Elements.

*Semantics*

$\mathfrak{M}_{\mathfrak{i}}[\![\texttt{true}]\!] \equiv \top$ *(the meaning of* ***true*** *is customary)*
$\mathfrak{M}_{\mathfrak{i}}[\![\texttt{false}]\!] \equiv \bot$ *(the meaning of* ***false*** *is customary)*

### Annex Y.3.1.1 Property Constant

(1) Property constants are values that are defined in AADL property sets.[2]

```
property_constant ::=
  property_set_identifier :: property_constant_identifier
```

*Semantics*

(S1) The meaning of property constants are defined by the AADL standard, AS5506B §11.1.3 Property Constants.

[1] BA D.7(4)
[2] AS5506B §11.1.3 Property Constants

### Annex Y.3.1.2 Property Reference

(1) Property values may be defined in property sets, or attached to a component or feature.[3]

```
property_reference ::=
  ( # [ property_set_identifier :: ]
  | component_element_reference #
  | unique_component_classifier_reference #
  | self )
  property_name
```

(2) The property may be relative to the component containing the behavior annex subclause: a subcomponent, a bound prototype, a feature, or the component itself.

```
component_element_reference ::=
  subcomponent_identifier | bound_prototype_identifier
  | feature_identifier | self
```

(3) Because AADL property values may be arrays or records, a property name may include array indices or record field identifiers.

(4) When the property is a range, the upper bound or lower bound of the property value can be referenced using `upper_bound` and `lower_bound` keywords.[4]

(5) When a property is a record, the field of a property value can be referenced using a dot separator between the property identifier and the field identifier.[5]

(6) When a property is an array, elements of the property value can be referenced using an integer value between brackets.[6]

```
property_name ::= property_identifier { property_field }*

property_field ::= [ integer_value ] | . field_identifier
  | . upper_bound | . lower_bound
```

(7) Property values may be from any component specified by its package name, type identifier, and optionally implementation identifier.

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

## Annex Y.3.2 Assertion Name

(1) An *assertion name* is a sequence of identifiers, with optional array indices, separated by periods. Section §**??**, Types, defines the relationship between names and elements of values having constructed types:

[3] Assertion Differs from BA: no local variable properties
[4] BA D.7(9)
[5] BA D.7(10)
[6] BA D.7(11)

arrays, records, and variants. A slice, or portion of an array, may be named by an integer-valued range as its array index.

```
assertion_name ::= root_identifier { [ index_expression_or_range ] }*
   { . field_identifier  { [ index_expression_or_range ] }*  }*
```

(2) An array index must be an integer expression, or a *slice* defined as an integer-valued range: lower bound . . upper bound.

```
index_expression_or_range ::=
     integer_expression [ ..  integer_expression ]
```

*Legality Rules*

(L1) Array indices must be non-negative.

(L2) An array index or slice must be in the array's range. Names with array indexes outside of the array's range have undefined value and have undefined type.

(L3) A slice's lower bound must be at most its upper bound.

*Semantics*

(S1) Where $x$ is a variable name,[7] $y$ is a value, $s$ is a state, and the pair $(x, y) \in s$:

$\mathfrak{M}_s[\![x]\!] \equiv y$ *(the meaning of a variable name in a state is its value)*

Where $a$ is an array name, $i$ is an integer value or values for a multidimensional array, $y$ is a value, $s$ is a state, and the pair $(a[i], y) \in s$:

$\mathfrak{M}_s[\![a[i]]\!] \equiv y$ *(the meaning of an array in a state is the value associated with its index)*

Where $r$ is an record name, $l$ is a label, $y$ is a value, $s$ is a state, and the pair $(r.l, y) \in s$:

$\mathfrak{M}_s[\![r.l]\!] \equiv y$ *(the meaning of a record in a state is its value of its selected label)*

Where $v$ is a variant name with discriminator $d$, $l$ is a label, $y$ is a value, $s$ is a state, and the pairs $(v.d, l), (v.l, y) \in s$:

$\mathfrak{M}_s[\![v.l]\!] \equiv y$ *(the meaning of a variant is the value of the element having the label of the discriminator)*

# Annex Y.3.3 Port Value

(1) The core language defines that data from data ports is made available to the application source code through a port variable having the name of the port. If no new value is available since the previous freeze, the previous value remains available and the variable is marked as not fresh. Freshness can be tested in the application source code via service calls [AS5506B §8.3.5].[8]

```
port_value ::= in_port_name ( ? | 'count | 'fresh | 'updated  )
```

[7] A name may be a simple identifier, or a compound name using indexes and/or labels. Here that name must correspond to a variable. In the following the name must correspond to an array, record or variant.

[8] Assertion Differs from BA: port names must have suffix: ? or '

```
port_name ::=
  { subcomponent_identifier . }* port_identifier
    [ [ natural_literal ] ]
```

# Chapter Y.4

# Lexicon

(1) Numeric literals, whitespace, identifiers and comments follow AS5506B §15 Lexical Elements.[1] String literals are enclosed in ` ' like LaTeX.

## Annex Y.4.1 Character Set

(1) The only characters allowed outside of comments are the graphic_characters and format_effectors.

```
character ::= graphic_character | format_effector
    | other_control_character

graphic_character ::= identifier_letter | digit | space_character
     | special_character
```

(2) The character repertoire for the text of BLESS annex libraries, subclauses, and properties consists of the collection of characters called the Basic Multilingual Plane (BMP) of the ISO 10646 Universal Multiple-Octet Coded Character Set, plus a set of format_effectors and, in comments only, a set of other_control_functions; the coded representation for these characters is implementation defined (it need not be a representation defined within ISO-10646-1).

(3) The description of the language definition of BLESS uses the graphic symbols defined for Row00: Basic Latin and Row 00: Latin-1 Supplement of the ISO 10646 BMP; these correspond to the graphic symbols of ISO 8859-1 (Latin-1); no graphic symbols are used in this standard for characters outside of Row 00 of the BMP. The actual set of graphic symbols used by an implementation for the visual representation of the text of BLESS is not specified.

(4) The categories of characters are defined as follows:

```
identifier_letter
   upper_case_identifier_letter | lower_case_identifier_letter
```

[1]BA D.7(6)

```
upper_case_identifier_letter
   Any character of Row 00 of ISO 10646 BMP whose name begins
  Latin Capital Letter.
lower_case_identifier_letter
  Any character of Row 00 of ISO 10646 BMP whose name begins
  Latin Small Letter.
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
space_character
  The character of ISO 10646 BMP named Space.
special_character
  Any character of the ISO 10646 BMP that is not reserved for a control
  function, and is not the space_character, an identifier_letter,
  or a digit.
format_effector
  The control functions of ISO 6429 called character tabulation (HT),
  line tabulation (VT),  carriage return (CR), line feed (LF), and
  form feed (FF).
other_control_character
  Any control character, other than a format_effector, that is allowed
  in a comment; the set of other_control_functions allowed in comments
  is implementation defined.
```

(5) Table Y.4.1 defines names of certain special_characters.

Table Y.4.1: Special Character Names

| Symbol | Name | Symbol | Name |
|---|---|---|---|
| " | quotation mark | # | number sign |
| = | equals sign | _ | underline |
| + | plus sign | , | comma |
| − | minus | . | dot |
| : | colon | ; | semicolon |
| ( | left parenthesis | ) | right parenthesis |
| [ | left square bracket | ] | right square bracket |
| { | left curly bracket | } | right curly bracket |
| & | ampersand | ^ | caret |

# Annex Y.4.2 Lexical Elements, Separators, and Delimiters

(1) The text of BLESS annex libraries, subclauses, and properties consist of a sequence of separate lexical elements. Each lexical element is formed from a sequence of characters, and is either a delimiter, an identifier, a reserved word, a numeric_literal, a character_literal, a string_literal, or a comment.

The meaning of BLESS annex libraries, subclauses, and properties depends only on the particular sequences of lexical elements that form its compilations, excluding comments.

(2) The text of BLESS annex libraries, subclauses, and properties are divided into lines. In general, the representation for an end of line is implementation defined. However, a sequence of one or more format_effectors other than character tabulation (HT) signifies at least one end of line.

(3) In some cases an explicit *separator* is required to separate adjacent lexical elements. A separator is any of a space character, a format_effector, or the end of a line, as follows:

- A space character is a separator except within a comment, or a string_literal.
- Character tabulation (HT) is a separator except within a comment.
- The end of a line is always a separator.

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

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

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

```
:= <> != :: => -> .. -[ ]-> )~>
```

(5) The following names are used when referring to compound delimiters:

| Delimiter | Name |
|---|---|
| := | assign |
| <> != | unequal |
| :: | qualified name separator |
| => | association |
| -> | implication |
| -[ | left step bracket |
| ]-> | right step bracket |
| )~> | right conditional bracket |

## Annex Y.4.3 Identifiers

(1) Identifiers are used as names. Identifiers are case sensitive.[2]

```
identifier ::= identifier_letter {[_] letter_or_digit}*
letter_or_digit ::= identifier_letter | digit
```

- An identifier shall not be a reserved word in either BLESS or AADL.
- Identifiers do not contain spaces, or other whitespace characters.

[2] Identifiers in AADL are case insensitive.

# Annex Y.4.4 Numeric Literals

(1) There are four kinds of *numeric literal*: integer, real, complex, and rational. A *real literal* is a numeric literal that includes a point, and possibly an exponent; an *integer literal* is a numeric literal without a point; a *complex literal* is a pair of real literals separated by a colon; a *rational literal* is a pair of integer literals separated by a bar.

(2) Peculiarly, negative numbers cannot be represented as numeric literals. Instead unary minus preceding a numeric literal represents negative literals instead.

```
numeric_literal ::=
  integer_literal | real_literal | rational_literal | complex_literal
```

(3) Integer values are equivalent to `Base_Types::Integer` values as defined in the AADL Data Modeling Annex B.[3]

```
integer_literal ::= decimal_integer_literal | based_integer_literal

real_literal ::= decimal_real_literal
```

## Annex Y.4.4.1 Decimal Literals

(1) A decimal literal is a numeric_literal in the conventional decimal notation (that is, the base is ten).

```
decimal_integer_literal ::= numeral

decimal_real_literal ::= numeral . numeral [ exponent ]

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

exponent ::= (E|e) [+] numeral | (E|e) - numeral
```

(2) An underline character in a numeral does not affect its meaning. The letter E of an exponent can be written either in lower case or in upper case, with the same meaning.

(3) An exponent indicates the power of ten by which the value of the decimal literal without the exponent is to be multiplied to obtain the value of the decimal literal with the exponent.

## Annex Y.4.4.2 Based Literals

(1) A based literal is a numeric_literal expressed in a form that specifies the base explicitly.

```
based_integer_literal ::= base # based_numeral # [ positive_exponent ]

base ::= digit [ digit ]

based_numeral ::= extended_digit [_] extended_digit

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

[3] BA D.7(7)

(2) The base (the numeric value of the decimal numeral preceding the first #) shall be at least two and at most sixteen. The extended_digits A through F represent the digits ten through fifteen respectively. The value of each extended_digit of a based_literal shall be less than the base.

(3) The conventional meaning of based notation is assumed. An exponent indicates the power of the base by which the value of the based literal without the exponent is to be multiplied to obtain the value of the based literal with the exponent. The base and the exponent, if any, are in decimal notation. The extended_digits A through F can be written either in lower case or in upper case, with the same meaning.

### Annex Y.4.4.3 Rational Literals

A *rational literal* is the ratio of two integers.

```
rational_literal ::=
  [ [-] dividend_integer_literal | [-] divisor_integer_literal ]
```

### Annex Y.4.4.4 Complex Literals

A *complex literal* is a pair of real numbers for the real part and imaginary part.

```
complex_literal ::=
  [ [-] real_literal : [-] imaginary_part_real_literal ]
```

## Annex Y.4.5 String Literals

(1) A string_literal is formed by a sequence of graphic characters (possibly none) enclosed between two string brackets: ` and '.[4]

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

string_element ::= "" | non_string_bracket_graphic_character
```

(2) The sequence of characters of a string literal is formed from the sequence of string elements between the string bracket characters, in the given order, with a string element that is "" becoming " in the sequence of characters, and any other string element being reproduced in the sequence.

(3) A null string literal is a string literal with no string elements between the string bracket characters.

## Annex Y.4.6 Comments

(1) A comment starts with two adjacent hyphens and extends up to the end of the line. A comment may appear on any line of a program.

[4]BLESS string literals are different from AADL string literals which use " as string bracket characters.

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

(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.

Chapter Y.5

# Alphabetized Grammar

```
actual_assertion_parameter ::=
  formal_identifier : actual_assertion_expression                    §Y.2.4.6 p22

actual_assertion_parameter_list ::=
  actual_assertion_parameter { , actual_assertion_parameter }*       §Y.2.3.5 p14

assertion ::=
  << ( assertion_predicate
  | assertion_function
  | assertion_enumeration
  | assertion_enumeration_invocation ) >>                            §Y.2.2 p7

assertion_annex_library ::=
  annex Assertion {** { assertion }+ **} ;                           §Y.2.1 p6

assertion_enumeration ::=
  assertion_enumeration_label_identifier : parameter_identifier
  +=> enumeration_pair { , enumeration_pair }*                       §Y.2.2.4 p9

assertion_enumeration_invocation ::=
  +=> asserion_enumeration_label_identifier
  ( actual_assertion_parameter )                                     §Y.2.4.7 p23
```

```
assertion_expression ::=
  assertion_subexpression
    [ { + assertion_subexpression }+
    | { * assertion_subexpression }+
    | - assertion_subexpression
    | / assertion_subexpression
    | ** assertion_subexpression
    | mod assertion_subexpression
    | rem assertion_subexpression ]
  | sum logic_variables [ logic_variable_domain ]
      of assertion_expression
  | product logic_variables [ logic_variable_domain ]
      of assertion_expression
  | numberof logic_variables [ logic_variable_domain ]
      that subpredicate                                          §Y.2.4 p18
assertion_function ::=
  [ label_identifier : [ formal_assertion_parameter_list ] ]
   := ( assertion_expression | conditional_assertion_function )   §Y.2.2.3 p9
assertion_function_invocation ::=
  assertion_function_identifier ( [ assertion_expression |
  actual_assertion_parameter { , actual_assertion_parameter }* ] )  §Y.2.4.6 p22
assertion_predicate ::=
  [ label_identifier : [ formal_assertion_parameter_list ] : ]
  predicate                                                      §Y.2.2.2 p8
assertion_range ::=
  assertion_subexpression range_symbol assertion_subexpression   §Y.2.3.6 p15
assertion_subexpression ::=
  [ - | abs ]  timed_expression
  | assertion_type_conversion                                    §Y.2.4 p18
assertion_type_conversion ::=
  ( natural | integer | rational | real | complex | time )
  parenthesized_assertion_expression                             §Y.2.4 p19
assertion_value ::=
  now | tops | timeout
  | value_constant
  | variable_name
  | assertion_function_invocation
  | port_value                                                   §Y.2.4.3 p20
component_element_reference ::=
  subcomponent_identifier | bound_prototype_identifier
  | feature_identifier | self                                    §Y.3.1.2 p26
conditional_assertion_expression ::=
  ( predicate ?? assertion_expression : assertion_expression )   §Y.2.4.4 p20
```

```
conditional_assertion_function ::=
    condition_value_pair { , condition_value_pair }*                          §Y.2.4.5 p21

condition_value_pair ::=
    parenthesized_predicate -> assertion_expression                           §Y.2.4.5 p21

enumeration_pair ::= enumeration_literal_identifier -> predicate              §Y.2.2.4 p9

event ::= < port_variable_or_state_identifier >                               §Y.2.3.10 p17

event_expression ::=
  [not] event
  | event_subexpression (and event_subexpression)+
  | event_subexpression (or event_subexpression)+
  | event - event                                                             §Y.2.3.10 p17

event_subexpression ::=
  [ always | never ] ( event_expression ) | event                             §Y.2.3.10 p17

existential_quantification ::=
   exists logic_variables logic_variable_domain
   that predicate                                                             §Y.2.3.9 p16

formal_assertion_parameter ::= parameter_identifier [ ~ type_name ]           §Y.2.2.1 p7

formal_assertion_parameter_list ::=
  formal_assertion_parameter { , formal_assertion_parameter }*                §Y.2.2.1 p7

index_expression_or_range ::=
    integer_expression [ .. integer_expression ]                              §Y.3.2 p27

integer_expression ::=
  [ - ]
  ( integer_assertion_value
  | ( integer_expression - integer_expression )
  | ( integer_expression / integer_expression )
  | ( integer_expression { + integer_expression }+ )
  | ( integer_expression { * integer_expression }+ ) )                        §Y.2.3.4 p13

logic_variable_domain ::=
  in ( assertion_expression range_symbol assertion_expression
    | predicate )                                                             §Y.2.3.8 p16

logic_variables ::=
  logic_variable_identifier { , logic_variable_identifier }*
  : type                                                                      §Y.2.3.8 p16
name ::=
  root_identifier { [ index_expression_or_range ] }*
    { . field_identifier { [ index_expression_or_range ] }* }*                §?? p??

parenthesized_assertion_expression ::=
  ( assertion_expression )
  | conditional_assertion_expression
  | record_term                                                               §Y.2.4.2 p20
```

```
parenthesized_predicate ::= ( predicate )                                §Y.2.3.7 p15
port_name ::=
  { subcomponent_identifier . }* port_identifier
    [ [ natural_literal ] ]                                              §Y.3.3 p28
port_value ::=
  in_port_name ( ? | 'count | 'fresh | 'updated )                        §Y.3.3 p27
predicate ::=
  universal_quantification
  | existential_quantification
  | subpredicate
    [ { and subpredicate }+
    | { or subpredicate }+
    | { xor subpredicate }+
    | implies subpredicate
    | iff subpredicate
    | -> subpredicate ]                                                  §Y.2.3 p10
predicate_invocation ::=
  assertion_identifier
   ( [ assertion_expression | actual_assertion_parameter_list ] )        §Y.2.3.5 p14
predicate_relation ::=
  assertion_subexpression relation_symbol assertion_subexpression
  | assertion_subexpression in assertion_range
  | shared_integer_name += assertion_subexpression                       §Y.2.3.6 p15
property_constant ::=
  property_set_identifier :: property_constant_identifier                §Y.3.1.1 p25
property_field ::=
  [ integer_value ]
  | . field_identifier
  | . upper_bound
  | . lower_bound                                                        §Y.3.1.2 p26
property_name ::= property_identifier { property_field }*                §Y.3.1.2 p26
property_reference ::=
  ( # [ property_set_identifier :: ]
  | component_element_reference #
  | unique_component_classifier_reference #
  | self # )
  property_name                                                          §Y.3.1.2 p26
range_symbol ::= .. | ,. | ., | ,,                                       §Y.2.3.6 p15
relation_symbol ::= = | < | > | <= | >= | != | <>                        §Y.2.3.6 p15
```

```
subpredicate ::=
  [ not ]
  ( true | false | stop
  | predicate_relation
  | timed_predicate
  | event_expression
  | def logic_variable_identifier )                                   §Y.2.3.1 p11

time_expression ::=
  time_subexpression
  | time_subexpression - time_subexpression
  | time_subexpression / time_subexpression
  | time_subexpression { + time_subexpression }+
  | time_subexpression { * time_subexpression }+                     §Y.2.3.3 p12

time_subexpression ::= [ - ]
  ( time_assertion_value
  | ( time_expression )
  | assertion_function_invocation )                                  §Y.2.3.3 p12

timed_expression ::=
    ( assertion_value
      | parenthesized_assertion_expression
      | predicate_invocation )
    [ ' | ^ integer_expression | @ time_expression ]                 §Y.2.4.1 p19

timed_predicate ::=
  ( name | parenthesized_predicate | predicate_invocation )
  [ ' | @ time_expression | ^ integer_expression ]                   §Y.2.3.2 p11

type_name ::=
  { package_identifier :: }* data_component_identifier
    [ . implementation_identifier ]
  | natural | integer | rational | real
  | complex | time | string                                          §Y.2.2.1 p8

unique_component_classifier_reference ::=
  { package_identifier :: }* component_type_identifier
  [ . component_implementation_identifier ]                          §Y.3.1.2 p26

universal_quantification ::=
  all logic_variables logic_variable_domain
  are predicate                                                      §Y.2.3.8 p16

value_constant ::=
  true | false | numeric_literal | string_literal
  | property_constant | property_reference                           §Y.3.1 p25
```

**Alphabetized Lexicon**

```
base ::= digit [ digit ]                                             §Y.4.4.2 p32

based_integer_literal ::= base # based_numeral # [ positive_exponent ]   §Y.4.4.2 p32
```

```
based_numeral ::= extended_digit [_] extended_digit                                §Y.4.4.2 p32
character ::= graphic_character | format_effector
    | other_control_character                                                      §Y.4.1 p29
comment ::= --{non_end_of_line_character}*                                         §Y.4.6 p33
complex_literal ::=
  [  [-] real_literal :  [-] imaginary_part_real_literal  ]                        §Y.4.4.4 p33
decimal_integer_literal ::= numeral                                                §Y.4.4.1 p32
decimal_real_literal ::= numeral . numeral [ exponent ]                            §Y.4.4.1 p32
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9                                    §Y.4.1 p30
exponent ::= (E|e) [+] numeral | (E|e) - numeral                                   §Y.4.4.1 p32
extended_digit ::= digit | A | B | C | D | E | F | a | b | c | d | e | f           §Y.4.4.2 p32
format_effector
  The control functions of ISO 6429 called character tabulation (HT),
  line tabulation (VT), carriage return (CR), line feed (LF), and
  form feed (FF).                                                                  §Y.4.1 p30
graphic_character ::= identifier_letter | digit | space_character
     | special_character                                                           §Y.4.1 p29
identifier ::= identifier_letter {[_] letter_or_digit}*                            §Y.4.3 p31
identifier_letter
   upper_case_identifier_letter | lower_case_identifier_letter                     §Y.4.1 p29
integer_literal ::= decimal_integer_literal | based_integer_literal                §Y.4.4 p32
letter_or_digit ::= identifier_letter | digit                                      §Y.4.3 p31
lower_case_identifier_letter
  Any character of Row 00 of ISO 10646 BMP whose name begins
  Latin Small Letter.                                                              §Y.4.1 p30
numeral ::= digit {[_] digit}*                                                     §?? p??
numeric_literal ::=
  integer_literal | real_literal | rational_literal | complex_literal              §Y.4.4 p32
other_control_character
  Any control character, other than a format_effector, that is allowed
  in a comment; the set of other_control_functions allowed in comments
  is implementation defined.                                                       §Y.4.1 p30
rational_literal ::=
  [ [-] dividend_integer_literal |  [-] divisor_integer_literal  ]                  §Y.4.4.3 p33
real_literal ::= decimal_real_literal                                              §Y.4.4 p32
space_character
  The character of ISO 10646 BMP named Space.                                      §Y.4.1 p30
```

```
special_character
  Any character of the ISO 10646 BMP that is not reserved for a control
  function, and is not the space_character, an identifier_letter,
  or a digit.
```
§Y.4.1 p30

```
string_element ::= "" | non_string_bracket_graphic_character
```
§Y.4.5 p33

```
string_literal ::= "{string_element}*"
```
§Y.4.5 p33

```
upper_case_identifier_letter
   Any character of Row 00 of ISO 10646 BMP whose name begins
  Latin Capital Letter.
```
§Y.4.1 p29

# Index