Annex Document Z
Behavior Language for Embedded Systems with Software

Normative

v0.9

May 1, 2016

0

Note to SAE International AS-2C AADL Standard Committee members:

This draft v0.9 has annotations in color and footnotes, that will be omitted from the balloted draft, especially for the Committee.

Direct quotes from Etienne's BA-with-errata revision of the original BA text going for informal ballot, are colored like this.

Parts concerning the merged formal semantics of BLESS with JP's synchronous semantics, are colored like this, and are indexed under 'JP'.

Quotations of BA, generate both footnotes and index entries under "BA quotation". References to BA paragraphs, of similar subject, but not quoted, get footnote and listings under "BA quotation", too, but are not colored.

Items related to BLESS reconciliation get footnotes and listing in the index under "Reconciliation".

Some of the significant differences between BA and BLESS get footnotes and listing in the index under "BLESS Differs from BA".

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.[1]

(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.[2]

(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

[1] BA intro (1)
[2] BA intro (2)

implementations against systems models throughout the development lifecycle.[3]

(4) The original Annex D document defined a behavior *modeling* language that became known as 'BA' for Behavior Annex which targeted 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 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.

[4]

(5) BA was meant to abstractly model behavior for schedulability analysis, and the like, early in the system design process. The actual design artifact (i.e. Ada program) would be written by hand, albeit with some help from code generators to make sure program interfaces conform to architecture. Having the actual design artifacts, better estimates could be used to re-run the analyses, but the defined BA behavior would never be used in a product. Lack of a type system or semantics was no hindrance to such behavior modeling.

(6) Concurrently, work began on a state-transition language that would have grammar as close to BA as possible, but diametrically opposite intent. The Behavior Language for Embedded Systems with Software (BLESS) would exactly represent behavior, that could be compiled down to machine code, precisely enough so formal methods could be used to verify that the executed code would conform to its specification. Consequently, a type system was created for to be consistent with both the AADL property types, and the AADL Data Modeling Annex (AS5506/2 Annex B). Every production in BLESS was formally defined. Assertions could be interspersed with behavior to be a proof outline, and used as a formal behavior interface specification language. Thus BLESS began to diverge from BA.

(7) BLESS reconciles BA with BLESS, but also formal semantics for BA devised by Jean-Pierre Talpin for reasoning about system timing. Upon close inspection, it was realized that both systems of formal semantics defined different aspects of the same thing, at different levels of abstraction. Merger of semantics required (mostly) changing labels and representation. *The formal semantics defined herein are not specific or restricted to any particular tools or formal methods, providing guidance as to meaning of BLESS text such that other formal methods and tools can be applied consistently.*

(8) BLESS specifications and behaviors can be attached to AADL models using an *annex subclause*. If applied to component type specifications, an annex subclause applies to all the associated implementations. If a component is extended, annex subclauses defined in an ancestor are applied to its descendants except when the later defines its own annex subclause of the same kind.

(9) An annex subclause can be specified for a specific *mode* by appending an `in modes` clause.[5] If the annex subclause is not mode specific, then it must be unique and it applies for all modes. If no mode-specific annex subclauses apply to a mode, then the behavior is undefined.[6] The foregoing applies to

[3] BA intro (3)
[4] BA scope (1)
[5] AS5506B §12 Modes and Mode Transitions
[6] Best never to have behavior undefined in possible modes; explicitly state is does nothing in modes when not active.

all AADL annex sublanguages, not just the three defined by this document.

(10) The arrival of events and event data on ports of a non-periodic thread is an external trigger for dispatching the thread; it initiates a transition defined in the AADL core standard[7]. A transmission request (a.k.a. event) on an outgoing port is an external trigger to the state transition system of a virtual bus or bus. Dispatch conditions are specified in terms of external triggers via event port, event data port, or time out. Dispatch does not depend on the input value, which can only impact the action following the dispatch.

(11) The `Behavior_Specification` of a component may access a shared data component made accessible through a requires data access feature. The current data value of such shared data component available to the `Behavior_Specification` is determined at the time of access.

(12) Grammar productions follow AS5506B with the exception of literal symbols.[8] The standard way of writing literals in **`bold`** works fine for reserved words, but can be hard to see for symbols. To make literal symbols in grammar productions easier to see, they have been colored **`purple`**. Listings of examples have reserved words from AADL in **`red`**, and those new to BLESS in **`blue`**. There are a few exceptions, and some of the special symbols in BLESS are also **`blue`**.

[7] AS5506B §5.4.1
[8] AS5506B §1.5 Method of Description and Syntax Notation

Contents

List of Figures

List of Tables

Chapter Z.1

Scope

(1) This document defines Architecture Analysis and Design Language (AADL) annex sublanguages to allow behavior specifications to be attached to AADL components, superseding the previous Annex D, collectively called 'BAv2':

- a reactive, state machine language called `Behavior_Specification`, and
- a programming language for subprograms called `Subprogram`.

(2) `Behavior_Specification` is meant to be an AADL annex sublanguage for *thread* and *device* components that interact with other components through events on ports, and have indefinite lifetimes.

(3) `Subprogram` is meant to be an AADL annex sublanguage for *subprogram* components that are passively invoked with input parameters, execute for a finite duration, returning output parameters upon termination.

(4) Both `Behavior_Specification` and `Subprogram` allow insertion of optional non-executable assertions defined in Chapter Z.9 Assertion. Assertions may be used as a behavior interface specification language (BISL) or to express what is true about the program (or system) in states, or at points of execution. Formal semantics are defined as Hoare triples: if I know P is true (precondition), and I do S, then Q will be true (postcondition). P and Q would be assertions, while S would be some action defined in Chapter Z.6 Action.

This document is organized into

Annex Z.1: Scope this chapter

Annex Z.2: Overview of Behavior Annex Concepts provides background and conceptual overview

Annex Z.3: Behavior Specification defines the syntax and semantics of the **state automaton** used for the `Behavior_Specification` annex subclause.[1]

[1] BA D.1(4)

Annex Z.4: Thread Dispatch 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.[2]

Annex Z.5: Component Interaction 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.[3]

Annex Z.6: Action defines the syntax and semantics of the **action language** used to specify the transition actions of the automaton.[4]

Annex Z.7: Expression defines the syntax and semantics of the **expression language** used in the various parts of the behavior annex.[5]

Annex Z.8: Type defines the syntax and semantics of the **type language** which may be used in variable declarations in lieu of data component names defined using the Data Modeling language.

Annex Z.9: Assertion defines the syntax and semantics of the **assertion language** which allows formal statements of what is true about the system when events or data are received or issued by ports, threads occupy states, during performance of an action, or always true (invariant) of a component.

Annex Z.10: Subprogram defines the restrictions of the action and expression languages for behaviors of subprograms defined in `Subprogram` annex subclauses.

Appendix 1: Mathematics introduces all the mathematical concepts and notation used in this document.

Appendix 2: Lexicon defines the language elements used in this document, which are the same as the core AADL in AS5506B.

Appendix ??: Language Subsets defines useful subsets of BAv2 for various purposes.

Appendix 3: Package and Properties describes the predeclared property sets and package defined for BAv2.

Appendix 4: Alphabetized Grammar provides a list of all grammar productions, in alphabetical order, with links to where the production is defined.

[2] BA D.1(4)
[3] BA D.1(4)
[4] BA D.1(4)
[5] BA D.1(4)

Chapter Z.2

Overview of BLESS Concepts

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

(2) The state transition system of a `Behavior_Specification` consists of a collection of states and transitions between the states. The state automaton has one initial state, from which the automaton behavior starts. The state automaton also has one (or more) final states. When this state is reached the behavior is considered to have completed. The state automaton can have complete states that represent temporary suspension of execution and resumption based on external trigger conditions. Finally, the state automaton can have discrete states of execution behavior.[2]

(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.[3]

(4) The behavior of a component such as a sampling periodic thread or a thread processing commands, may start in an initial state; initialize itself and suspend itself at a complete state; reactivate from the complete state repeatedly based on time or the arrival of an external event or message; transitioning from the initial state to the first complete state, or between complete states may involve transitioning to intermediate computational states; a termination request results in a transition to a final state.[4]

[1] BA D.2(1)
[2] BA D.2(2)
[3] BA D.2(3)
[4] BA D.2(4)

(5) The transitions of a state transition system specify behavior as a change of the current state from a source state to a destination state. A condition determines whether a transition is taken, and an action is performed when the condition evaluates to true. Transition priorities control the evaluation order to transition conditions, thus, the transition to be taken if the conditions of multiple transitions hold; otherwise the transition choice is non- deterministic. Transition conditions fall into two categories: conditions that affect the execution of a thread based on external triggers (dispatch conditions); and conditions that model behavior within an execution sequence of a thread, subprogram or other component (execution conditions) and are based on input values from ports, shared data, parameters, and behavior variable values.[5]

(6) The *timed dispatch protocol* of the core AADL standard specifies a time out condition for dispatches relative to the previous dispatch. The `Behavior_Specification` allows the modeler to define dispatch time out conditions relative to the previous completion as well as time out conditions on blocks of behavior actions. Furthermore, the `Behavior_Specification` allows modelers to specify what behavior is to occur when such time outs occur.[6]

(7) 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]

(8) 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[8] as well as the state transition system of the `Behavior_Specification` of the thread. The transmission request on an outgoing port is an external trigger to the state transition system of a virtual bus or bus. Dispatch conditions are specified in terms of external triggers via event port, event data port, calls received on provides subprogram access features, or time out. Dispatch does not depend on the input value, which can only impact the action following the dispatch.[9]

(9) External trigger conditions, consumption of input and generation of output must be consistent with the input/output semantics of the core AADL standard. For example, if the core model specifies a subset of the ports to be external dispatch triggers, then the external trigger condition can only specify conditions on this subset. Similarly, additional ports can be specified in both the core model and in the `Behavior_Specification` to indicate that the port content is frozen at dispatch time. These two specifications must be consistent.[10]

(10) Input on ports is frozen according to the semantics of the core AADL standard[11] 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`.[12]

[5] BA D.2(5)
[6] BA D.2(6)
[7] BA D.2(7)
[8] AS5506B §5.4.1 Thread States and Actions
[9] BA D.2(8)
[10] BA D.2(9)
[11] AS5506B §8.3.2 Port Input and Output Timing
[12] BA D.2(10)

(11) The `Behavior_Specification` of a component may access a shared data component made accessible through a requires data access feature. The current data value of such shared data component available to the `Behavior_Specification` is determined at the time of access.[13]

[13] BA D.2(11)

Chapter Z.3

Behavior Specification

(1) Behavior specifications can be attached to any AADL component types and component implementations using an annex subclause[1] with label `Behavior_Specification`.[2]

```
annex Behavior_Specification {**  . . .  **};
```

(2) 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 subclauses defined in the ancestor are applied to the descendent except if the later defines its own behavior annex subclause.[3] However, AS5506B §5.4 Threads, defines standard behavior for thread scheduling and interaction. Any component with a behavior specification must conform to the standard for threads, regardless of its component classifier. Therefore, references to 'thread' should be considered applicable to any component with a behavior specification annex subclause.

(3) A behavior annex subclause may be interpreted as a refinement of a call sequence section in a thread or subprogram component implementation. If both a call sequence section and a behavior annex subclause with subprogram call actions are defined for the same component implementation, then all the subprogram calls specified in the former must be reflected in the latter, although the call order may differ.[4][5]

(4) Mode-specific behavior by appending an annex subclause with an **in modes** clause.[6] Alternatively, 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.[7]

[1] BA D.3(1)
[2] Implementations may also accept annex labels `BAv2` or `BLESS` equivalently.
[3] BA D.3(1)
[4] BA D.3(22)
[5] **Reconciliation:** call sequence
[6] AS5506B §12 Modes and Mode Transitions
[7] BA D.3(3)

Z.3.1 Component Behavior

(1) Component behavior is defined by a *state transition system*. A state transition system has a set of states, a set of local variables some of which may have initial values, and a set of transitions. The transitions of a state transition system specify behavior as a change of the current state from a source state to a destination state.

(2) Component behaviors may have an *assert clause* listing labelled assertions to be used by other assertions.

(3) Component behaviors may have an *invariant clause* that must be true of every state.

```
behavior_annex ::=
    [ assert { assertion }+ ]
    [ invariant assertion ]
    [ variables ]
    states { behavior_state }+
    [ transitions ]
```

Legality Rule

(L1) Component behaviors must have at least two states (`initial` and `final`) and at least one transition.[8]

Naming Rule

(N1) The variable, state, and transition identifiers must be unique within an annex subclause, and may not also be data subcomponents, component features, or mode identifiers–except for complete state identifiers which may be mode identifiers.[9]

Consistency Rules

(C1) If a component type or implementation is extended, behavior specification defined in the ancestor are applied to the descendent except if the later defines its own behavior specification.[10]

(C2) A behavior specification of a subcomponent overrides the behavior specification of its containing component if they conflict.[11]

Semantics

(S1) A `Behavior_Sepcification` defines an automaton (Appendix 1.19), A, as behavior for the component (usually a thread) which contains it in an annex subclause, $A = (S_A, s_0, V_A, P_A, T_A, C_A)$. Its states, S_A are defined in the `states` section having unique initial state s_0. Its persistent variables, V_A are defined in the `variables` section. Its ports, P_A are defined by its containing component. Its transitions, T_A are defined in the `transitions` section. Its constraints, C_A, are defined in . . . , denoted by multi-sorted logical formula F_A.[12]

[8] BA D.3(L1)
[9] BA D.3(N1)
[10] BA D.3(C1)
[11] BA D.3(C2)
[12] Huh?

Z.3.2 Behavior States

(1) The `states` section declares all the states of the automaton. Some states may be qualified as `initial` state, `final` state, or `complete` state, or combinations thereof. A state without qualification will be referred to as execution state. A behavior automaton starts from an initial state and terminates in a final state.[13] A behavior state may have an assertion that holds when that state is current.

(2) The core AADL standard defines runtime execution states for threads.[14] These states include an *initial* state (thread halted), a *complete* state (awaiting dispatch,) and a *final* state (stopped thread).

```
behavior_state ::=
    behavior_state_identifier
      : [initial] [complete] [final] state [ assertion ] ;[15]
```

(3) 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. 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.[16]

(4) In the case of subprograms, the automaton consists of one `initial` state representing the starting point of a call, zero or more intermediate execution states, and one `final` state. A `final` state represents the completion of a call. The complete state is not used in behavior specifications of subprograms.[17]

(5) When a component has modes it may also have a separate behavior annex subclause for each mode. In this case, a mode transition results in a transition from the `complete` state of the current mode behavior automaton to the `initial` state of the behavior automaton of the new mode.[18]

(6) At least one state must be labeled `final`. There may be no transitions from a `final` state (unless it's also a `complete` state). The `final` state may be entered via a normal transition, abort transition, stop transition, or invocation of the component's *Finalize_Entrypoint*.[19] 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

[13] BA D.3(8)
[14] AS5506B §5.4.1 Thread States and Actions
[15] BLESS Differs from BA: Only a single state identifier is allowed.
[16] BA D.3(24)
[17] BA D.3(9)
[18] BA D.3(13)
[19] AS5506B §5.4.1 Thread States and Actions

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.[20]

(7) Entering a `complete` state suspends the component until its next dispatch. 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.[21]

(8) Execution states are transitory allowing computations upon dispatch to be subdivided into steps. From every execute state there must be at least one transition leaving that state with an enabled transition condition. Upon dispatch, a finite number of execute states may occur before entering a complete or final state.

(9) Upon completion of initialization a `complete` state is reached starting from the `initial` state, and perhaps a finite number of execution states.

(10) 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.[22] This means that no transition can reach the implicit initial state.

(11) An `initial` state can be a `complete` state and a `final` state as well (i.e. an final complete state). Such a state is an implicit superposition of three states an `initial` state, a `complete` state, and a `final` state - connected by two implicit transitions. The first transition, from the implicit initial state towards the implicit complete state, can only be triggered by the initialization action as defined in the core standard. The second transition, from the implicit complete state and towards the implicit final state, can only be triggered by the reception of a stop event. Note that exiting (respectively entering) an initial final complete state stands for exiting (resp. entering) the implicit complete state. No other action than the initialization action (call to the initialize_entrypoint as defined by a property in the core language) can be associated to the first implicit transition. No other action than the finalization action (represented by the finalize_entrypoint property from the core language) can be associated to the second implicit transition. No execution condition can be associated to those two implicit transitions.[23]

Legality Rules

(L1) A Behavior_Specification component behavior annex specification must define one `initial` state. A Behavior_Specification component behavior annex specification must define at least one `final` state.[24]

[20] BA D.3(12)
[21] BA D.3(12)
[22] BA D.4(7)
[23] BA D.4(8)
[24] BA D.3(L1)

(L2) Transitions from an execute source, have execute conditions, which are boolean expressions evaluated by the component.

(L3) Transitions from a `complete` source, have dispatch conditions, evaluated by the AADL runtime services. [25]

(L4) Transitions from states that are `final` only (not also `complete` or `initial`) are not allowed.[26]

(L5) A behavior annex specification for a thread, device, and other components awaiting dispatch or awaiting a mode transition, must define at least one complete state and one initial state. This may be the same state.[27]

(L6) A behavior annex specification for threads and other components with initialization and finalization entrypoints may explicitly model the initialization and finalization by including one initial state and one or more final states.[28]

(L7) A behavior annex specification for a subprogram must not define any complete states.[29]

Consistency Rules

(C1) A `Behavior_Specification` for a thread must be consistent with the core AADL semantics.[30]

(C2) 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.[31]

(C3) A behavior annex subclause of a *subcomponent* overrides the behavior annex subclause of its containing component if they conflict.[32]

(C4) The behavior annex state transition system must not remain blocked in an *execution* state. This means that the logical disjunction of all the execute conditions associated with the transitions out of an execution state must be true.[33]

(C5) If the behavior annex defines transitions from a `complete` state that represents a *mode* in the containing component, then the transition condition associated with these transitions must be consistent with the corresponding mode transition triggers.[34][35]

(C6) In behavior transitions, *mode conditions* can be used to describe mode transitions in any component classifier, except those belonging to the category of threads and subprograms. In components of these categories, execute conditions and/or dispatch conditions should be used to describe behavior transitions.[36]

Semantics

[25] BA D.3(L6), BA D.3(L7)
[26] BA D.3(L8)
[27] BA D.3(L3)
[28] BA D.3(L4)
[29] BA D.3(L2)
[30] AS5506B §5.4 Threads
[31] BA D.3(C1)
[32] BA D.3(C2)
[33] BA D.3(C3)
[34] AS5506B §12 Modes and Mode Transitions
[35] BA D.3(C4)
[36] BA D.3(C5)

(S1) Entering a `complete` suspends execution until next dispatch, and sends all pending outputs.

(S2) Where S_t is the behavior state of the component at time t, i is a satisfying interval, s is a behavior state, d is a dispatch condition, and A is an assertion:

$\mathfrak{M}_{\text{i}}[\![$ `s initial state;` $]\!] \equiv S_{start(\text{i})} = s$
(the initial state is the state at the start of the interval)
$\mathfrak{M}_{\text{i}}[\![$ `s final state;` $]\!] \equiv S_{end(\text{i})} = s$
(the final state is the state at the end of the interval)
$\mathfrak{M}_{\text{i}}[\![$ `s complete state;` $]\!] \equiv \forall t \in \text{i} \mid (S_t = s) \rightarrow \neg\mathfrak{M}_t[\![d]\!] \wedge suspended(t)$
(for all time, component is suspended and the dispatch condition is false when in a complete state)
$\mathfrak{M}_{\text{i}}[\![$ `s <<A>> state;` $]\!] \equiv \forall t \in \text{i} \mid (S_t = s) \rightarrow \mathfrak{M}_t[\![A]\!]$
(for all time, when in a state, its assertion is true)

Example

```
states
  start : initial state;
  fill : complete state
    <<SpO2_INV() and (num_samples<#PulseOx_Properties::Num_Trending_Samples)>>;
  check : state;
  run : complete state;
  halt : final state;   --normal termination
  fail : final state;   --error termination
```

Z.3.3 Variables

(1) A `variables` clause declares identifiers that represent either local *behavior variables* in the scope of the current annex subclause, or a reference to an external data component. 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.[37] Values of variables *are* persistent across the various invocations of the same behavior annex subclause.[38] [39]

```
variables ::= variables { behavior_variable }+

behavior_variable ::= local_variable_declarator
    { ,  local_variable_declarator }* :
    type [ := value_constant ] [ assertion ] ;

declarator ::= identifier { array_size }*

array_size ::= [ natural_value_constant  ]
```

[37] BA D.3(6)
[38] BLESS Differs from BA: variable persistence
[39] BLESS Differs from BA: variables have no property associations

(2) Behavior variables retain *persistent* values retained between dispatches. Variables that retain state when the system is powered off are *nonvolatile*. Variables oxymoronically-declared to be *constant* may not be assigned except during initialization. Targets of combinable operations must be declared *shared*. Arrays whose concurrent access is controlled using combinable operations are declared *spread*, as in spread across memory banks to minimize bank conflict on concurrent accesses.[40] Variables that may only be assigned once are labeled *final*.

(3) Behavior variable declarations can indicate that a requires data access is `shared`. Only shared variables may be targets of combinable operations.

Legality Rules

(L1) Variables may have initialization expressions.

(L2) Referenced external data components must be `requires data access` features of the component.[41]

(L3) Variables labeled `final` may only be assigned once.

(L4) Variables labeled `constant` may not be assigned values except by their declaration.

Semantics

(S1) Where v is a behavior variable identifier, T is a type, e is an expression, and d is a data component identifier:

$\mathfrak{M}[\![$`variables v:T;`$]\!] \equiv \exists v \in T$ *(there exists a variable v of type T)*

$\mathfrak{M}[\![$`variables v:T:=e;`$]\!] \equiv \exists v \in T \wedge \mathfrak{M}_{start(i)}[\![v]\!] = \mathfrak{M}[\![e]\!]$
(there exists a variable v of type T with a value of e at the beginning of the interval)

Example

```
variables
  nts : constant integer:=#PulseOx_Properties::Num_Trending_Samples;
  spo2 :  array [1 ..nts] of PulseOx_Types::SpO2:=0;  --holds SpO2 history
  spo2_nxt : array [1 ..nts] of PulseOx_Types::SpO2:=0;
  num_samples :  integer:=0;  --counts samples while filling
```

Z.3.4 Transitions

(1) In a `Behavior_Specification`, *transitions* define dynamic behavior. When the component's current state is one of the source states of a particular transition, and the condition for transition evaluates to true, the current state will become the destination state after an action (if supplied) is performed. A transition's Assertion, if supplied, is invariant during the transition.

[40] If you're not seeking speed-up of computation via concurrent execution, you won't need `shared` or `spread`.
[41] AS5506B §8.6 Data Component Access

(2) A transition may be identified by a *label*. The label contains a transition identifier and an optional priority number. Transition priorities control the evaluation order of transition guards.[42] The evaluation order of two transitions with the same priority is non-deterministic. Transitions with no specified priority have the lowest priority.[43]

(3) Actions can be performed by a transition before entry of the destination state. If a transition is enabled, the actions are performed and then the state specified as the destination of the transition becomes the new current state.[44] [45]

```
transitions ::= transitions { behavior_transition }+

behavior_transition ::=
    [ behavior_transition_label : ]
    source_state_identifier { , source_state_identifier }*
    -[ [ transition_condition ] ]-> destination_state_identifier
   [ { [ behavior_actions ] } ] [ assertion ] ;

behavior_transition_label ::=
    transition_identifier [ [ priority_natural_literal ] ]

transition_condition ::=
  dispatch_condition | execute_condition
  | mode_condition | internal_condition
```

(4) When the source state of a transition is a state where the component is waiting for dispatch, and if its dispatch protocol is not periodic, then the condition is a *dispatch condition* that specifies the triggering events in terms of event port, event data port, calls received on provides subprogram access features, or time out. Otherwise, when the source state is an execute state of the component, the condition is an *execute condition* on state variables and received input values.

(5) The core AADL standard defines dispatch conditions for threads in terms of a disjunction of trigger conditions as result of arrival of events or event data on incoming ports of subprogram access features. A subset of ports involved in the triggering of a dispatch may be specified through the `Dispatch_Trigger` property. The behavior specification can refine this dispatch condition into a Boolean condition that is associated with a transition out of a complete state.[46]

(6) A dispatch trigger may result in a transition out of a `complete` state and to one of the states defined in the `Behavior_Specification` (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 (periodic dispatch or timeout). Reaching a `complete` state can be interpreted as calling the `Await_Dispatch` run-time service. Thus a component is suspended if it performs a transition to a complete state, after having executed the action associated to the transition. The next dispatch will restart the thread from that state.[47]

[42] **Reconciliation:** transition priority
[43] BA D.3(19)
[44] **Reconciliation:** behavior action block
[45] BA D.3(20)
[46] BA D.3(26)
[47] BA D.3(27)

(7) When the `Dispatch_Protocol` property is timed or hybrid, the value of the time out dispatch condition is given by the `Period` property of the component.[48]

(8) An empty transition condition is equivalent to a condition that is always true.[49]

Legality Rule

(L1) A behavior specification for threads and other components must have one initial state and one or more final states.[50]

(L2) Behavior transitions having Assertions must have labels.

(L3) Transitions from states that are final only are not allowed.[51]

(L4) A behavior 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.[52]

Consistency Rules

(C1) The state transition system must not remain blocked in an execution state. This means that the logical disjunction of all the execute conditions associated with the transitions out of an execution state must be true.[53]

(C2) If the behavior specification defines transitions from a complete state that represents a mode in the containing component, then the `transition_condition` associated with these transitions must be consistent with the corresponding `mode_transition_triggers`.[54][55]

Semantics

(S1) A `behavior_transition` defines multiple transitions $(s, V_s, I_A, g, d, V_d, O_A, f) \in T_A$ of automaton A, because there are many possible input values, variable valuations, and output values for a transition from the source state to the destination state (Annex 1.19). The transition only occurs when the automaton occupies the source state `transition_condition` is true, which may be an execute condition if the source state is an execution state, or a dispatch condition if the source state is a complete state.

(S2) The `behavior_transition_label`, if present, defines a label, m, which represents the clock (Annex1.17) for the transition, $\hat{m}$, when the transition occurs. For transition `m:s-[g]-d;`, its clock is $\hat{m} \Leftrightarrow (s$ **and** $g)$.

Example

```
transitions
  sptt0: start-[ ]->fill{};
  sptt1: fill-[on dispatch]->check { . . . };
```

[48] BA D.3(28)
[49] BA D.3(N2)
[50] BA D.3(L3)
[51] BA D.3(L8)
[52] BA D.3(L8)
[53] BA D.3(C3)
[54] BA D.3(C3)
[55] AS5506B 12 Modes and Mode Transitions

```
sptt2a: check-[num_samples<#PulseOx_Properties::Num_Trending_Samples]->fill
  {(spo2',num_samples':=spo2,num_samples)};
sptt2b: check-[num_samples=#PulseOx_Properties::Num_Trending_Samples]->run
  {(spo2',num_samples':=spo2,num_samples)};
sptt2c: check-[num_samples>#PulseOx_Properties::Num_Trending_Samples]->fail{};
```

Z.3.5 Execute Condition

(1) Any transition leaving an execute state must have an *execute condition*.[56] Execute conditions are boolean expressions that may only contain references visible within the component, such as ports and local variables.

```
execute_condition ::=
  boolean_expression_or_relation | timeout | otherwise
```

Legality Rule

(L1) Any transition with an execute state as its source must have an execute condition, or nothing which is the same as `true`.

Semantics

(S1) Where s and d are behavior states in S with Assertions A_s and A_d,

```
states . . . s:state <<As>>; d:state <<Ad>>; . . .
```

$S_{start(\mathrm{i})}$ is the behavior state at time $start(\mathrm{i})$, $S_{end(\mathrm{i})}$ is the behavior state at time $end(\mathrm{i})$, b is a behavior condition, w is an asserted action, C is an Assertion, and i is a satisfying interval:

$$\mathfrak{M}_{\mathrm{i}}[\![\texttt{transitions s-[b]->d}]\!] \equiv \begin{array}{l} S_{start(\mathrm{i})} = s, \\ S_{end(\mathrm{i})} = d, \\ \mathfrak{M}_{start(\mathrm{i})}[\![A_s \wedge b]\!] \rightarrow \mathfrak{M}_{end(\mathrm{i})}[\![A_d]\!] \end{array}$$

(a transition from s to d on condition b over a subinterval i, must start in s with A_s, end in d with A_d, and the conjunction of the condition b and A_s at the beginning, must imply A_d at the end)

$$\mathfrak{M}_{\mathrm{i}}[\![\texttt{transitions s-[b]->d \{w\}}]\!] \equiv \begin{array}{l} S_{start(\mathrm{i})} = s, \\ S_{end(\mathrm{i})} = d, \\ \mathfrak{M}_{start(\mathrm{i})}[\![A_s \wedge b]\!] \rightarrow \mathrm{wp}(w, \mathfrak{M}_{end(\mathrm{i})}[\![A_d]\!]) \end{array}$$

(a transition from s to d on condition b with action w over a subinterval i, must start in s with A_s, end in d with A_d, and the conjunction of the condition b and A_s at the beginning, must imply the weakest precondition of w and A_d at the end)

$$\mathfrak{M}_{\mathrm{i}}[\![\texttt{transitions s-[b]->d \{w\} <<C>>}]\!] \equiv \begin{array}{l} S_{start(\mathrm{i})} = s, \\ S_{end(\mathrm{i})} = d, \\ \mathfrak{M}_{start(\mathrm{i})}[\![A_s \wedge b]\!] \rightarrow \mathrm{wp}(w, \mathfrak{M}_{end(\mathrm{i})}[\![A_d]\!]), \\ \mathfrak{M}_{\mathrm{i}}[\![C]\!] \end{array}$$

(a transition from s to d on condition b with action w and Assertion C over a subinterval i, must start in s

[56] BA D.3(18)

with A_s, end in d with A_d, and the conjunction of the condition b and A_s at the beginning, must imply the weakest precondition of w and A_d at the end; the Assertion C must be true throughout i *)*[57]

(S2) Semantics of in data and in event data ports are defined in §Z.5.4 and §Z.5.5 respectively.

Z.3.6 Internal Conditions

(1) Internal events may be used to represent interactions among annexes. In the scope of a behavior annex subclause, an internal feature may be used to describe under which circumstances an event is sent from either an internal event port or an internal event data port .[58]

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

Z.3.7 Modal Conditions

(1) The function `in mode` tests whether the current local mode is among the identifiers listed. The mode identifiers must be among those of the behavior annex subclauses in modes clause, if any, and the modes of its thread component.

(2) The `setmode` action initiates a mode switch to the identified mode, which must be among the modes of the thread.[59]

(3) When the state machine is used to define mode transitions, complete state identifiers match mode identifiers for the component. Leaving a mode-state requires a transition with a *mode condition* which may be triggered by an event (data) arriving or leaving an event (data) port of the component or one of its subcomponents.[60]

```
mode_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 )
subcomponent_port_reference ::=
  subcomponent_identifier { . subcomponent_identifier }* . port_identifier
```

[57] The Assertion in behavior transitions between the action and the terminating semicolon was changed from a post-condition to an invariant that holds during the transition. Previously, during execution of an action, a component was in *no* state. In no state, none of the state Assertions necessarily holds. This made it impossible to write a component invariant that was *always* true. By defining transitions' Assertions to hold during execution of its transition, the intrinsic component invariant becomes the disjunction of all state and transition Assertions.

[58] BA D.5(17)

[59] BLESS Differs from BA: `setmode`

[60] BLESS Differs from BA: mode trigger

```
logical_operator ::=
  and | or | xor | and then | or else
```

Naming Rule

(N1) If any complete state identifier is a mode identifier, then all complete state identifiers in that annex subclause must also be mode identifiers.

Consistency Rules

(C1) Modal behavior must conform to AS5506B §12, Modes and Mode Transitions.[61]

(C2) If transitions from a complete state that represents a mode in the containing component, then the behavior_condition associated with these transitions must be consistent with the corresponding `mode_transition_triggers` of a `mode_transition`.[62][63][64]

Z.3.8 Synchronization

(1) An automaton is said *well synchronized* iff all its transitions from one complete state to another can be performed using one big step (Annex 1.22). If all series of transitions from one complete state to another in an automaton do not use an output port twice or more, then the automaton is well synchronized.

(2) For a well-synchronized automaton, one can reduce the execution states introduced in the representation of action sequences by substituting variable names by their definitions in the formula that use them. For instance, $\{(s1, g1, s, v = u), (s, g2, s2, f2,)\}$ can be reduced as $\{(s1, g1 \wedge (g2[v/u]), s, v = u \wedge (f2[v/u]))\}$. One can also reduce action sequences and action sets by composing formulas representing independent actions. For instance, $\{(s1, g1, s, p1 = v1), (s, g2, s2, p2 = v2)\}$ can be reduced as $\{(s1, g1 \wedge g2, s2, p1 = v1 \wedge p2 = v2)\}$ iff $p1 \neq p2$, and so on. A well-synchronized automaton can be represented without execution states.

[61] BA D.3(C4)
[62] BA D.4(C4)
[63] AS5506B §12 Modes and Mode Transitions
[64] **Reconciliation:** mode

Chapter Z.4

Thread Dispatch

Z.4.1 Dispatch Condition

(1) Any transition leaving a `complete` state must have a *dispatch condition* that begins `on dispatch`. When a component has `Periodic` dispatch protocol, no *dispatch expression* is needed. For components with other dispatch protocols, a dispatch expression determines when a component is dispatched.

(2) A dispatch condition must be met to transition from a `complete` state.[1] A dispatch condition determines whether a transition is taken, and an action is performed when the condition evaluates to true. A dispatch condition is a Boolean-valued expression (disjunction of conjunctions of dispatch triggers) that specifies the logical combination of triggering events for the next dispatch. A *dispatch trigger* 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 timed event–either periodic dispatch or timeout). The ports used in the dispatch condition must be consistent with the ports listed in the core AADL model as dispatch triggers.

(3) A dispatch trigger can be the arrival of events or event data on ports, calls on provides subprogram access features, the stop event, and occurrence of dispatch related and completion related time outs.[2]

(4) Dispatch conditions must be evaluated by the run-time system, not the component, and must be insensitive to component state. Dispatch conditions must not depend upon which complete state is being resumed from, nor from persistent values of variables. Dispatch conditions must not consume events; dispatch conditions must decide solely on event's existence, not their data, nor queue depth. If no dispatch logical expression is supplied, dispatch occurs upon the default dispatch condition defined for the component's `Dispatch_Protocol`[3] property.

[1] BA D.4(2)
[2] BA D.4(3)
[3] AS5506B §A.2 Predeclared Thread Properties

(5) A dispatch condition may be absent (just `on dispatch`) indicating default dispatch at the end of the thread's period. Periodic dispatches are always considered to be implicit unconditional dispatch triggers on complete states and handled by dispatch conditions without dispatch trigger condition.[4]

(6) Dispatch conditions are evaluated to determine whether a dispatch occurs. If there are multiple outgoing transitions, the dispatch condition (if present) is evaluated to determine which transition is taken. If multiple transitions are eligible, then the priority value (Z.3.4) 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.[5]

(7) When the `Dispatch_Protocol` property is `Timed` or `Hybrid`, the value of the time out dispatch condition is given by the `Period` property of the component.[6]

```
dispatch_condition ::=
   on dispatch [ dispatch_expression ] [ frozen frozen_ports ]

dispatch_expression ::=
  dispatch_conjunction  { or dispatch_conjunction }*
  | stop
  | dispatch_relative_timeout_catch
  | completion_relative_timeout_catch
  | provides_subprogram_access_identifier

dispatch_conjunction ::=
   dispatch_trigger  { and dispatch_trigger }*

dispatch_trigger ::= in_event_port_name | in_event_data_port_name
  | port_event_timeout_catch
```

(8) A *dispatch trigger* is an event which causes the dispatch condition to be evaluated. The value of a dispatch condition is a boolean expression of dispatch triggers. Event arrival at either event ports or event data ports causes a dispatch trigger referenced by the port's identifier. The timeout dispatch trigger is covered in section Z.4.2 Timeout Dispatch Trigger.

(9) All *stop events* are dispatch triggers, caused by arrival of an event on the implicit *stop port*, to model initiation of finalization and transition from a complete state to the a state, possibly via one or more execution states. If the core property finalize entrypoint is already specified[7] then it can be used as an implicit finalization action, otherwise it can be specified as action on transitions from complete states.[8]

(10) The core AADL standard defines which ports are implicitly frozen at dispatch time, i.e., port that actually triggers a dispatch, or ports that do not trigger a dispatch. In the behavior annex subclause it is possible to explicitly specify as part of the dispatch condition a list of additional ports that must also be frozen although they do not take part to the dispatch condition. Otherwise, the port freeze action, `>>` can be used as a transition action.[9]

```
frozen_ports ::= in_port_name { , in_port_name }*
```

[4] BA D.4(4)
[5] BA D.3(27)
[6] BA D.3(28)
[7] AS5506B §5.4.1 Thread States and Actions
[8] BA D.4(6)
[9] BA D.4(1)

Naming Rules

(N1) The incoming port identifier in the frozen port list must refer to incoming ports in the component type to which the behavior annex subclause is associated.[10]

(N2) The incoming port identifiers and subprogram access feature identifiers that represent dispatch trigger events must refer to the respective feature in the component type to which the behavior annex subclause is associated.[11]

Legality Rules

(L1) The specification of frozen ports in the dispatch condition must be consistent with that of the core AADL model.[12] [13]

(L2) Table Z.4.1 sums up the compatibility rules between the dispatch_protocol property values defined in the core standard and the dispatch_trigger_condition used in a behavior annex. This table is only relevant when the property and the annex are applied to a component of the thread category.[14]

Table Z.4.1: Dispatch Protocol-Trigger Compatibility

`dispatch_trigger`	Periodic	Sporadic	Aperiodic	Hybrid	Timed
∅ (none)	X			X	X
`dispatch_expression`		X	X	X	X
Provides Subprogram Access		X	X	X	X
`stop`	X	X	X	X	X
`timeout` (only)					X

Consistency Rules

(C1) The specification of frozen ports in the dispatch condition must be consistent with that of the core AADL model.[15] [16]

Legality Rule

(L3) A behavior annex specification for a subprogram must not contain a dispatch condition in any of its transitions.[17]

Semantics

(S1) Where i is an interval, p is an input event port identifier, e is an event, S is a state (the start node of a satisfying lattice), and A, B, C, and D are dispatch triggers:

$\mathfrak{M}_S[\![A \textbf{ and } B]\!] \equiv \mathfrak{M}_S[\![A]\!] \wedge \mathfrak{M}_S[\![B]\!]$

(dispatch condition may be conjunction of dispatch triggers)

$\mathfrak{M}_S[\![(A \textbf{ and } B) \textbf{ or } (C \textbf{ and } D)]\!] \equiv (\mathfrak{M}_S[\![A]\!] \wedge \mathfrak{M}_S[\![B]\!]) \vee (\mathfrak{M}_S[\![C]\!] \wedge \mathfrak{M}_S[\![D]\!])$

[10] BA D.4(N1)
[11] BA D.4(N2)
[12] BA D.3(L9)
[13] AS5506B §5.4.8 Runtime Support For Threads
[14] BA D.4(L1)
[15] BA D.3(L9)
[16] AS5506B §5.4.8 Runtime Support For Threads (although this section says nothing about frozen ports)
[17] BA D.3(L5)

(dispatch condition may be disjunction of conjunctions of dispatch triggers)
$\mathfrak{M}_S[\![\mathrm{p}]\!] \equiv \exists e \in p$ *(the meaning of in event port identifier* `p` *is when an event exists at port p, it is a dispatch trigger)*

(S2) Execution of a transition occurs when the component is suspended in the transition's source state, its dispatch expression is true, and *no dispatch expression of transition leaving that state has been true since the time-of-previous-suspension* (**tops**). For a single transition R, leaving a complete state S, having dispatch expression D,

```
R: S -[on dispatch D ]-> . . .,
```

then R will be dispatched at time t:

$\mathfrak{M}_t[\![dispatch(R)]\!] \equiv \mathfrak{M}_t[\![S]\!] \wedge \mathfrak{M}_t[\![D]\!] \wedge \nexists\, t2 \in \{tops, , t\} \mid \mathfrak{M}_{t2}[\![D]\!]$
(transition R will be dispatched at time t when the component is in state S at time t, dispatch expression D is true at time t, and there was no time $t2$ since the time-of-previous-suspension $tops$ in which the component was dispatched)

(S3) When multiple transitions lease the same complete state, none of their dispatch conditions must be true since the time-of-previous suspension. For transitions R having dispatch expression D, $R2$ having dispatch expression $D2$, and $R3$ having dispatch expression $D3$, all having complete state S as source,

```
R: S -[on dispatch D ]-> . . .,
R2:S -[on dispatch D2 ]-> . . .,
R3:S -[on dispatch D3 ]-> . . .,
```

then R will be dispatched at time t:

$\mathfrak{M}_t[\![dispatch(R)]\!] \equiv \mathfrak{M}_t[\![S]\!] \wedge \mathfrak{M}_t[\![D]\!] \wedge \nexists\, t2 \in \{tops, , t\} \mid (\mathfrak{M}_{t2}[\![D \vee D2 \vee D3]\!])$
(transition R will be dispatched at time t when the component is in state S at time t, dispatch expression D is true at time t, and there was no time $t2$ since the time-of-previous-suspension $tops$ in which the component was dispatched for any transition leaving state S)

Z.4.2 Timeout Dispatch

(1) *Timeout* is a dispatch trigger that is raised after the specified amount of time since the last dispatch or the last completion is expired. In the `Timed` dispatch protocol, the `Timeout` property specifies the timeout value.[18]

```
dispatch_relative_timeout_catch ::= timeout
```

```
completion_relative_timeout_catch ::= timeout behavior_time
```

(2) Timeouts may include a list of event port identifiers, in or out, data or not. An event, in or out, on a port in the list resets and starts the timeout, regardless of component state. The component need not be in the source state of the transition having a timeout dispatch trigger to reset/start the timeout. A timeout dispatch trigger may include a port list. In this case, the behavior is as follows:[19][20]

[18] BA D.4(5)
[19] BA D.4(5)
[20] BLESS Differs from BA: timeout

- an event was received or sent by a listed port begins, or resets, the timeout interval
- if no event was received or sent by a listed port during the timeout interval, a dispatch trigger occurs

```
port_relative_timeout_catch ::=
  timeout ( port_identifier { [ or ] port_identifier }* )
    behavior_time
```

(3) A timeout dispatch trigger sans port list and behavior time is dispatch relative using the `Period` property for its duration.[21]

(4) Disjunction(`or`) of port names is optional.

Naming Rule

(N1) A port identifier refers to either an `in` port or an `in event` port.

Legality Rule

(L1) The `dispatch_relative_timeout_catch` condition must only be used for `Timed` threads, and must be declared in only one outgoing transition of a complete state.[22]

Semantics

(S1) Where p_1, p_2, and p_3 are event port identifiers, d is a duration that must either be a literal, or the name of an AADL property of type `Timing_Properties::Time`, and u is an `AADL_Properties::Time_Units` unit:

$$\mathfrak{M}_t[\![\texttt{timeout (p1 p2 p3) d u}]\!] \equiv \begin{array}{l} \mathfrak{M}_{now-d}[\![p_1 \vee p_2 \vee p_3]\!] \wedge \\ \nexists s \in \{now - d, , now\} \mid \mathfrak{M}_s[\![p_1 \vee p_2 \vee p_3]\!] \end{array}$$

(the meaning of `timeout` is an event arrived, or was issued, at one of the listed ports (p_1 p_2 or p_3), d time previously, and no events arrived, or were issued, at any of the listed ports since then)

Z.4.3 `abort` and `stop` events

(1) AS5506B defines semantics for `stop` and **`abort`** events.[23] A **`stop`** dispatch trigger occurs when a component is requested to enter its *component halted* state through a *stop* request after completing the execution of a dispatch or while not part of the active mode. In this case, the component may execute a Finalize_Entrypoint before entering the *component halted* state.[24]

(2) An `abort` dispatch trigger occurs through an *abort* request to cause the component to immediately enter the *component halted* state. For both `stop` and `abort` a `final` state will be entered, never to leave again. The difference is that `stop` executes a `Finalize_Entrypoint` to clean up before halting; that behavior is the action of the `stop` transition.[25]

[21] BA D.4(L2)
[22] BA D.4(L2)
[23] AS5506B §5.4 Threads, esp. Figure 5 Thread States and Actions.
[24] BA D.4(6)
[25] Both `stop` and `abort` will occur automatically, so only users that need to define some special behavior action at their occurrence will use them.

(3) In a behavior specification, a *final* state can be a complete state (i.e. a *final complete* state). Such a state is an implicit superposition of two states a *complete* state and a *final* state - connected by an implicit transition. This implicit transition from the implicit *complete* state towards the implicit *final* state can only be triggered by the reception of a `stop` event. No other action than the finalization action (represented by the `Finalize_Entrypoint` property from the core language) can be associated to this implicit transition. No execution condition can be associated to this implicit transition. Note that entering (respectively exiting) a final *complete* state stands for entering (resp. exiting) the implicit *complete* state.[26]

(4) In a behavior specification, an `initial` state can be a `complete` state and a `final` state as well (i.e. an `initial final complete state`). Such a state is an implicit superposition of three states an `initial` state, a `complete` state, and a `final` state - connected by two implicit transitions. The first transition, from the implicit `initial` state towards the implicit `complete` state, can only be triggered by the initialization action as defined in the core standard. The second transition, from the implicit `complete` state and towards the implicit `final` state, can only be triggered by the reception of a `stop` event. Note that exiting (respectively entering) an `initial final complete` state stands for exiting (resp. entering) the implicit `complete` state. No other action than the initialization action (call to the `Initialize_Entrypoint` as defined by a property in the core language) can be associated to the first implicit transition. No other action than the finalization action (represented by the `Finalize_Entrypoint` property from the core language) can be associated to the second implicit transition. No execution condition can be associated to those two implicit transitions.[27]

Naming Rules

(N1) The incoming port identifier in the frozen port list must refer to incoming ports in the component type to which the behavior annex subclause is associated.[28]

(N2) The incoming port identifiers and subprogram access feature identifiers that represent dispatch trigger events must refer to the respective feature in the component type to which the behavior annex subclause is associated.[29]

Legality Rules

(L1) `stop` transitions must have `final` states as destinations.

Semantics

(S1) $\mathfrak{M}_S[\![\textbf{stop}]\!] \equiv \exists e \in$ `stop` and there must be a sequence of zero or more, delay-free execute conditions before reaching a **`final`** state.
*(the meaning of **`stop`** is when an event exists at special port `stop`, it is a dispatch trigger)*
$\mathfrak{M}_S[\![\textbf{abort}]\!] \equiv$ immediate component halt
*(the meaning of **`abort`** is halt immediately)*[30]

Example

[26] BA D.4(7)
[27] BA D.4(8)
[28] BA D.4(N1)
[29] BA D.4(N2)
[30] AS5506B §5.4.1 Thread States and Actions (20) and Figure 5

(5) This example specifies that the component should be dispatched if either an event arrives at port a, or events have arrived for both ports c and d.

```
annex Behavior_Specification {**
  states S1,S2:state; S3,S4:final state;
  . . .
  transitions
  S1-[on dispatch a or (c and d)]->S2;
  S1-[stop]->S3 {finalize action} ;
  S2-[stop]->S3 {different finalize action} ;
  S1-[abort]->S4 ; --no action, S4 is final for abort
   . . .
**}
```

Z.4.4 Thread Providing Subprogram Dispatch

(1) Provides subprogram access features that are declared in a `thread` component type can act as a dispatch triggers. The values of incoming parameters, if any, can then be used by naming the parameter within the scope of the behavior annex.[31]

(2) The core AADL standard supports modeling of remote procedure calls through provides subprogram access features on threads. The arrival of a call acts as a dispatch trigger to the thread. Calls are queued if the thread has not completed a previous dispatch. By default the call is a synchronous call with the calling thread being blocked, which corresponds to a *synchronous* `Subprogram_Call_Type` property.[32] To specify non-blocking calls, a *semi-synchronous* `Subprogram_Call_Type` property must be applied to the subprogram.[33]

[31] BA D.5(20)
[32] AS5506A 5.2
[33] BA D.5(21)

Chapter Z.5

Component Interaction

(1) Threads can interact through shared data component implementations, connected ports and subprogram calls. The AADL execution model defines the way queued event/data of a port are transferred to the thread in order to be processed and when a component is dispatched.[1]

(2) Messages can be received by the component through declared features of the current component type. They can be in or in out data ports; in or in out event ports; in or in out event data ports and in or in out parameters of subprogram access. Event and event data ports are associated with queues.[2]

Z.5.1 Communication Action

(1) Communication actions provide interaction with other components. A *communication action* sends or receives values from ports.[3] Actions of `in` ports and `out` ports are covered in following sections.

(2) Actions on ports consist of the input freeze action (`p>>`), the initiate send action with or without value assignment (`p!(v)` or `p!`), and parameterless subprogram calls (`sub()`) or subprogram calls with parameters (`sub(f1:a1, f2:a2, f3:a3)`). Another form of component interaction is through reading and writing of shared data components, which is expressed by the assignment action.[4]

```
communication_action ::=
  subprogram_invocation
  | output_port_name ! [ ( expression ) ]
  | input_port_name ? ( target )
  | frozen_input_port_name >>
```

[1] BA D.5(1)
[2] BA D.5(2)
[3] Subprogram invocation is a basic action Z.6.4.
[4] BA D.6(10)

```
port_name ::=
  { subcomponent_identifier . }* port_identifier
    [ [ natural_literal ] ]
target ::= local_variable_name | output_port_name
  | data_component_reference
data_component_reference ::=
  data_subcomponent_name { . data_subcomponent_name }*
  | data_access_feature_name { . data_field }*
  | data_access_feature_prototype_name { . data_field }*
data_field ::=
  data_subcomponent_name
  | data_access_feature_name
  | data_access_feature_prototype_name
```

Semantics

(S1) Accessing data components outside of a thread break encapsulation of state, is therefore error-prone, and thus stridently discouraged.

Z.5.2 Freeze Port

(1) The core language defines that input on ports is determined by default frozen at dispatch time, or at a time specified by the `Input_Time` property[5] and initiated by a `Receive_Input` service call[6] in the source text. From that point in time the input of the port during this execution is not affected by arrival of new data, events, or event data until the next time input is frozen.[7] [8]

(2) Freezing of input port content during execution requires consistency between the `Input_Time` property in the core model and the freeze input action, `p>>`. Similarly, initiating transmission of port output must be consistent between the `Output_Time` property in the core model and the port output, `p!`.[9]

(3) Ports causing a dispatch event are implicitly frozen at the time specified by the `Input_Time` property if the property specifies a deterministic value. It is also possible to explicitly freeze additional ports if it is consistent with their `Input_Time` property. As long as it remains consistent with the `Input_Time` property of a port, an explicit call to the `Receive_Input` service can be performed thanks to the frozen statement of the dispatch condition. With the same consistency constraints with respect to the `Input_Time` as a transition action.[10]

Consistency Rules

[5] AS5506B §9.2.4 Port Communication Timing
[6] AS5506B §8.3.5 Runtime Support For Ports
[7] BA D.5(3)
[8] **Reconciliation:** >> freeze port
[9] BA D.5(6)
[10] BA D.5(7)

(C1) The specification of frozen ports in the dispatch condition must be consistent with that of the core AADL model.[11]

(C2) Freezing of input port content during execution requires consistency between the `Input_Time` property in the core model and the freeze input action (`p>>`) in the Behavior_Specification.[12] [13]

Semantics

(S1) An input freeze action `p>>` is represented by turning s into a `complete` state with $T(g,s,d)[\texttt{p>>}] = (s, g?p, true, d)$.[14]

Z.5.3 In Event Ports

(1) Communication actions do not refer to *in event port*(s). Instead, an event arriving at an event port is a dispatch trigger used in dispatch transition conditions leaving complete states. See Z.4.1 Dispatch Condition for thread response to events arriving at an `in event port`.

Z.5.4 In Data Ports

(1) An *in data port* holds the most recent value sent to it. The port value may be explicitly assigned to a local variable with a communication action, `p?(v)`, or used in an expression or execute transition conditions (§Z.7.1 Value).

(2) The core language defines that data from data ports is made available to the application source code (and `Behavior_Specification`) through a port variable with 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[15] and in the `Behavior_Specifcation` via functions.[16]

Table Z.5.1: In Data Port AADL Runtime Service Call

Meaning	Grammar	Corresponding service call
read value	`p?`	`Get_Value` and then `Next_Value`
read into variable	`p?(var)`	`Get_Value`

Semantics

(S1) For each `in data port`, r, in the context of interval (state lattice) i:[17]

[11] AS5506B §5.4.8
[12] BA D.5(6)
[13] BA D.5(C1)
[14] This doesn't seem right. No new dispatch, just the value doesn't change until completion. JP, please clarify.
[15] AS5506B §8.3.5 Runtime Support For Ports
[16] BA D.5(4)
[17] The interval i starts at Dispatch time and ends at Completion.

r?(v) $\mathfrak{M}_t[\![\text{r?(v)}]\!] \equiv \mathfrak{M}_t[\![\text{v}]\!] = \mathfrak{M}_t[\![\text{r}]\!]$
(the variable gets the value of the port at the instant of its evaluation, which because frozen is the value of the port at dispatch)

(S2) To get data from an in data port, a transition read the value of the port and assigns it to the target variable. $T(s, g, d)[r?(v)] = (s, g, c, v = r)$.

Z.5.5 In Event Data Ports

(1) The complexity of `in event data port` behavior comes from its buffer. Unlike plain `in data port` which reports the most recent value received, `in event data port` buffers all the values received while waiting for dispatch, if any. Thus, the need for `updated`, `count`, and `fresh` to monitor the input buffer.

Values of `in event data port` may be used in one of two ways:

p use current value in expression, don't dequeue

p? use current value in expression, dequeue

p?(v) assign current value to variable, dequeue

(2) The core language defines that input on ports is determined by default at dispatch time, or at a time specified by the `Input_Time` property[18] and initiated by a `Receive_Input` service call[19] in the source text.

(3) The core language defines that data from data ports are made available to the component in a port variable.[20] Freshness of this data can be tested as a condition, `p'fresh`.[21]

(4) Event and event data ports have queues and the queues are processed as follows according to the core standard.[22] [23]

- A `Dequeue_Protocol` of `OneItem` makes one item available in a port variable and removes it from the port queue. If the queue is empty the port variable content is considered not fresh.
- A `Dequeue_Protocol` of `AllItems` removes all items from the port queue and places them into a local port queue (local to the state transition system). The component input events of the transition condition and the input actions of the transition action consume data elements from this local port queue. Any data not consumed as part of a transition will be lost, when the local queue content is overwritten with new input at the next execution.
- The `Dequeue_Protocol` of `MultipleItems` determines the content of the port queue and makes it available through the local port queue. In this case elements are removed from the

[18] AS5506B §8.3.2 Port Input and Output Timing
[19] AS5506B §8.3.5 Runtime Support for Ports
[20] AS5506B §8.3.3 Port Queue Processing
[21] BA D.5(4)
[22] AS5506B §8.3.3 Port Queue Processing
[23] BA D.5(5)

queue as they are consumed. Any elements not consumed remain in the queue and become available at the next execution.

Table Z.5.2: In Event Data Port AADL Runtime Service Calls

Meaning	Grammar	Corresponding service call
freeze	`p>>`	`Receive_Input`
test of updated	`p'updated`	`Updated`
get unread count	`p'count`	`Get_Count`
test of freshness	`p'fresh`	`Get_Count` > 0
read	`p`	`Get_Value`
read and dequeue	`p?`	`Get_Value` and then `Next_Value`
read into variable and dequeue	`p?(var)`	`Get_Value` and then `Next_Value`

(5) Within a behavior annex subclause, the following constructs are available to get the status and the contents of an input port `p`:[24]

- `p` can be used as a value and returns the most-recent (unless frozen) data stored in the port variable if `p` is a non-empty data port or an event data port with the `OneItem Dequeue_Protocol` using the `Get_Value` runtime service. The value cannot overwritten if the port direction is `in out` by writing to it. It does not dequeue an event and `p'count` is not decremented. Applied to an empty data port, `p` returns `null`.[25]
- `p'count` is equivalent to a call to the `Get_Count` runtime service: `p'count` returns the number of elements available through the port variable. In the case of a data port its value is one, or zero if no new value was received. In the case of an event port or event data port it is the number of frozen elements. If it is strictly positive, `pcount` is decremented when an element is dequeued.[26]
- `p'updated` is equivalent to a call to the Updated runtime service. `pupdated` returns true if some new values were received in port `p` since the last freeze of the port. Note that this operator is used to represent a call to the `Updated` service defined in the core AADL standard. This operation is performed without freezing the port `p`.[27]
- `p'fresh` returns true if the port variable has been refreshed at the previous dispatch. `p'fresh` is equivalent to the expression `Get_Count > 0`. In the case of a data port, this means that it has received a new value by the previous dispatch or freeze. In the case of an event data port this means that one or more elements from the port queue were frozen and are available for processing by the `Behavior_Specification` through `p`. If the port queue is empty at freeze time or the `p?` operation is applied to a port variable with no remaining elements, the value is not considered fresh.[28]
- `p?` is equivalent to a call to the `Get_Value` and `Next_Value` runtime services: `p?` dequeues an event or event data from a non empty event port queue. If it is strictly positive, the value of

[24] BA D.5(9)
[25] **Reconciliation:** non-dequeued port
[26] BA D.5(9)
[27] BA D.5(9)
[28] BA D.5(9)

`p'count` is decremented. In the case of an event data port the new first element is available in the port variable.

- When used in a behavior action, `p?(v)` dequeues an event from a non-empty event port queue, returning its value, and `p'count` is decremented using the `Get_Value` and `Next_Value` run-time services. Each use of `p?(v)` dequeues another event, returns its value, and decrements `p'count`. Applied to an empty event data port queue, `p?(v)` causes exception with label `Read_Empty_Event_Data_Port` to be thrown.[29]
- `p>>` is equivalent to a call to the `Receive_Input` runtime service, freezing the value so that subsequent references to `p` in the same dispatch receive the same value.[30]

Semantics

(S1) For each `in event data port`, p,

p $\mathfrak{M}_i[\![p]\!]$ ≡ Get_Value *(peak at oldest value w/o removal, use AADL runtime service Get_Value[31])*

p? $\mathfrak{M}_i[\![p?]\!]$ ≡ Get_Value then Next_Value *(retrieve oldest value, decrement count use AADL runtime service Next_Value[32])*

p?(v) $\mathfrak{M}_i[\![p?(v)]\!]$ ≡ Get_Value then Next_Value *(retrieve oldest value, decrement count use AADL runtime service Next_Value[33])*

p'count $\mathfrak{M}_i[\![p'count]\!]$ ≡ Get_Count *(count of values queued, use AADL runtime service Get_Count[34])*

p'fresh $\mathfrak{M}_i[\![p'fresh]\!]$ ≡ Get_Count > 0 *(new port value, use AADL runtime service Updated[35])*

p'updated $\mathfrak{M}_i[\![p'updated]\!]$ ≡ Updated.FreshFlag *(new port value, use AADL runtime service Updated[36])*

Z.5.6 Concurrency Control

(1) Within a `Behavior_Specification` subclause the value of incoming parameters of the containing subprogram type is returned by the corresponding formal parameter identifier. The value of the parameter has been frozen at the time of the call. Multiple references to a formal parameter return the same value. In the case of subprogram calls, outgoing parameters return their result by assigning them to the local variable named as the corresponding formal parameter identifier. The local variable can then be referenced to return its value.[37]

[29] BLESS Differs from BA: empty dequeue exception
[30] By default, port values are frozen at dispatch.
[31] AS5506B §8.3.5 Runtime Support for Ports (50) Get_Value
[32] AS5506B §8.3.5 Runtime Support for Ports (52) Next_Value
[33] AS5506B §8.3.5 Runtime Support for Ports (52) Next_Value
[34] AS5506B §8.3.5 Runtime Support for Ports (51) Get_Count
[35] AS5506B §8.3.5 Runtime Support for Ports (53) Updated
[36] AS5506B §8.3.5 Runtime Support for Ports (53) Updated
[37] BA D.5(11)

(2) Access to shared data subcomponents is controlled according to the `Concurrency_Control_Protocol` property specified associated with this data subcomponent.[38] If concurrency control is enabled, critical sections boundaries are defined by one of the following ways:

- By explicit definition of the time range over which a set of referenced shared data subcomponent are accessed. This is done using the `*!<` for starting time (resp. `*!>`) for ending time) locking actions (see Z.6.12 Locking Actions) within a behavior action block. If the critical section contains references to several shared data subcomponents, then resource locking will be done in the same order as the occurrence of the references to the shared data subcomponents and resource unlocking will be done in the reverse order. These operators may be used to refine the value of the `Access_Time` property[39] if it has been specified.
- By calls to appropriate provides subprogram access of the corresponding data component that have been explicitly defined to implement the concurrency control protocol.
- By explicit calls to the `Get_Resource` (resp. `Release_Resource`) runtime service[40] that can be achieved using the `!<` (resp. `!>`) operators applied to the shared data subcomponent identifier.

[41]

(3) A transition action can write data values to a shared data component by naming the data component directly or the data access identifiers declared in the component type on the left-hand side of an assignment, i.e., `d := v`.[42]

Z.5.7 Out Ports

(1) Messages can be sent within a `Behavior_Specification` subclause through declared features of the current component type. They can be: `out` or `in out` data ports; `out` or `in out` event ports; `out` or `in out` event data ports.[43]

(2) The sending of messages is consistent with the timing semantics of the core language. The core language specifies the output time through an `Output_Time`[44] property and the sending of the output is initiated by a `Send_Output` service call in the source text. For data ports the output is implicitly initiated at completion time (or deadline in the case of delayed data port connections).[45]

(3) Within a `Behavior_Specification` subclause, the following constructs are available to set the value of an out port p:[46].

- `p!` calls `Put_Value` on an event or event data port. The event is sent to the destination with assigned data, if any, according to the `Output_Time` property.

[38] AS5506B §5.1 Data
[39] AS5506B §8.6 Data Component Access
[40] AS5506B §5.1.1 Runtime Support For Shared Data Access
[41] BA D.5(12)
[42] BA D.5(16)
[43] BA D.5(13)
[44] AS5506B §8.3.2 Port Input and Output Timing
[45] BA D.5(14)
[46] BA D.5(15)

Table Z.5.3: Out Communication Actions

Meaning	Grammar	Corresponding service call
put	`p :=v`	`Put_Value`
send	`p!`	`Send_Output`
put and send	`p!(v)`	`Put_Value` and then `Send_Output`

- `p := d` calls `Send_Output` data or event data port. Data is transferred to the destination port according to the `Output_Time` property.
- `p!(d)` writes data `d` to the event data port `p` and calls the `Put_Value` and then `Output_Time` services. The event is sent to the destination according to the `Output_Time` property.

Consistency Rule

(C1) If the sending time of an output port is specified with the `Output_Time` property[47], then no send output action must be specified in the corresponding behavior actions of the `Behavior_Specification` subclause, or the two statements must be equivalent.[48]

Semantics

(S1) The precondition of port output must imply the assertion property of the port. The conjunction of the precondition of port output and event occurrence must imply the postcondition of port output.
Rule (Event Port Output)**.**

$$\textit{[EPOut]}\ \frac{A \wedge p@now \rightarrow B \qquad A \rightarrow \mathfrak{M}[\![p]\!]}{\ll A \gg\ p!\ \ll B \gg}$$

(NEED TO ADD INFERENCE RULES FOR DATA AND EVENT DATA PORTS)

(S2) For each `out event port`, p,

$\mathfrak{M}_i[\![p!]\!] \equiv$ Put_Value()
(send event from port, use AADL runtime service Put_Value with no DataValue parameter[49]; issued at `Output_Time`)

(S3) Equivalently, to send an out port event, a transition causes the clock of the port to be true.
$T(s,g,d)[p!] = (s,g,d,\hat{p})$.

(S4) For each `out event data port`, p,

$\mathfrak{M}_i[\![p!(e)]\!] \equiv \mathfrak{M}_i[\![p{:=}e]\!] \equiv$ Put_Value(e)
(send event data from port, use AADL runtime service Put_Value; issued at `Output_Time`)

(S5) Equivalently, to send data on an out event data port, a transition causes the value of the port to be the data sent, and then reset.
$T(s,g,d)[p!(e)] = \{(s,g,c,p=e)(c,g,d,\neg\hat{p})\}$ by introducing a new execution state c.

[47] AS5506B 8.3.2
[48] BA D.5(C2)
[49] AS5506B §8.3.5 Runtime Support for Ports (45) Put_Value

(S6) To send data on an out data port, a transition causes the value of the port to be the data sent, but not reset.
$T(s,g,d)[p!(e)] = T(s,g,d)[p := e] = (s,g,c,p = e)$.

Z.5.8 Subprogram Invocation

(1) Subprograms may be invoked (called) as an action.

```
  subprogram_invocation ::= subprogram_name ( [parameter_list] )
subprogram_name ::= subprogram_prototype_name
  | required_subprogram_access_name | subprogram_subcomponent_name
  | subprogram_unique_component_classifier_reference
  | required_data_access_name . provided_subprogram_access_name
  | local_variable_name . provided_subprogram_access_name
parameter_list ::= parameter { , parameter }*
parameter ::= [ formal_parameter_identifier : ] actual_parameter
actual_parameter ::= target | expression
```

(2) Requires subprogram access features that are declared in the component type can be called inside an action block of a transition. Call parameters can be previously received subprogram parameters, ports value or assigned temporary variables.[50]

(3) Subprogram invocations can be used in sequential composition or concurrent composition. In sequential composition they represent synchronous subprogram calls, while in concurrent composition they represent semi-synchronous calls, i.e., multiple calls are initiated to be performed simultaneously. The concurrent composition is considered to have completed when all subprogram calls within that set have completed. Note that the results from out parameters of one simultaneous call cannot be used as input to another call or other action in the same concurrent composition.[51]

(4) Within Behavior_Specification subclauses, the following constructs are available to call required subprogram `s`:[52]

- `s()` calls the parameter-less subprogram referred to by the requires subprogram access feature `s`.
- `s(f1:a1,...,fn:an)` calls the subprogram referred to by the requires subprogram access feature `s` with the corresponding formal-actual parameter list.[53]

(5) By default the subprogram invocation is synchronous with the calling thread being blocked, which corresponds to a `Synchronous Subprogram_Call_Type` property.[54] To specify non-blocking calls, a `SemiSynchronous Subprogram_Call_Type` property must be applied to the subprogram.[55]

[50] BA D.5(18)
[51] BA D.6(14)
[52] BA D.5(19)
[53] BLESS Differs from BA: formal-actual subprogram parameters
[54] AS5506B §5.2 Subprograms and Subprogram Calls
[55] BA D.5(21)

Legality Rule

(L1) In a subprogram invocation, the parameter list must match the signature of the subprogram being invoked.[56]

Semantics

(S1) The weakest-precondition semantics of subprogram invocation are defined by substituting actual parameters for formal parameters in the subprogram's pre- and post-conditions.
WP (Subprogram Invocation **[SI]**). *Where* $pre|^{f_1,\ldots,f_k}_{a_1,\ldots,a_k}$ *and* $post|^{f_1,\ldots,f_k}_{a_1,\ldots,a_k}$ *are the precondition and postcondition of p after actual parameters are substituted for formal parameters:* $\mathrm{wp}(p(f_1 : a_1, \ldots, f_k : a_k), Q) \equiv Q|^{pre}_{post}$

Where p is a subprogram name and $f_1 : a_1, \ldots, f_k : a_k$ are formal-actual parameter pairs:

$\mathfrak{M}_{\underline{i}}[\![p(f_1 : a_1, \ldots, f_k : a_k)]\!] \equiv \mathfrak{M}_{\underline{i}}[\![p|^{f_1,\ldots,f_k}_{a_1,\ldots,a_k}]\!]$

(S2) Subprogram invocations are specified using the communication protocols `HSER`, `LSER` or `ASER` defined in (TBD).[57] A subprogram invocation is hence translated by the composition of the client (the caller) and server (the callee) with the behavior of the calling protocol.

[56] BA D.6(L5)
[57] Wherever D.8 Synchronization Protocols are to be defined

Chapter Z.6

Action

(1) Actions associated with transitions are action blocks that are built from basic actions and a minimal set of control structures allowing action sequences, action sets, conditionals and finite loops. Action sequences are executed in order, while actions in actions sets can be executed in any order. Finite loops allow iterations over finite integer ranges.[1]

Z.6.1 Behavior Actions

(1) The *behavior actions* may be a single asserted action (Z.6.2), sequential composition of actions (Z.6.5), or concurrent composition of actions (Z.6.6).

```
behavior_actions ::=
   asserted_action
   | sequential_composition
   | concurrent_composition
```

Z.6.2 Asserted Action

(1) An *asserted action* is an action that may have assertions as pre- and post-conditions.[2] No terminating semicolon occurs after the post-condition. Semicolon is used for sequential composition.

```
asserted_action ::=
   [ precondition_assertion ]
   action
   [ postcondition_assertion ]
```

[1] BA D.6(1)
[2] BLESS Differs from BA: asserted action

Semantics

(S1) Where P and Q are predicates, and S is an action:

$\mathfrak{M}_{\underline{i}}[\![\ll P \gg \ S \ \ll Q \gg]\!] \equiv \mathfrak{M}_{start(\underline{i})}[\![P]\!] \wedge \mathfrak{M}_{end(\underline{i})}[\![Q]\!] \wedge \mathfrak{M}_{\underline{i}}[\![S]\!]$

(the meaning of subprogram behavior is that P is true in the stating state of i, Q is true in the ending state of i, and i satisfies S)

Inference Rule

(S2) An asserted action $\ll P \gg \ S \ \ll Q \gg$ is true, if `P` implies the *weakest precondition* (wp) of `S` and `Q`.

$$\textsc{Weakest Precondition:}\ \textbf{[WP]}\ \frac{P \rightarrow \mathrm{wp}(S, Q)}{\ll P \gg \ S \ \ll Q \gg}$$

(S3) Equivalently, $\ll P \gg \ S \ \ll Q \gg$ has the behavior of an automata transition $T(s, true, d, true)[S]$ from state s in which assertion $\ll P \gg$ holds, to state d in which assertion $\ll Q \gg$ holds while performing action S.

Example

```
<<INVW() and (PCA_Properties::Drug_Library_Size=k)>>
No_Drug_Found!    --indicate drug code not found
 <<DL()>>
```

Z.6.3 Action

(1) An *action* may be a basic action (Z.6.4), an alternative formula (Z.6.7), a loop (Z.6.10), a for-all (Z.6.9), a locking action (Z.6.12) or a block (Z.6.8).

```
action ::=
   basic_action
   | behavior_action_block
   | alternative
   | for_loop
   | forall_action
   | while_loop
   | do_until_loop
   | locking_action
```

Z.6.4 Basic Actions

(1) Basic actions can be assignment actions, communication actions or time consuming actions[3], or no action at all (`skip`). Threads can perform actions forbidden for subprograms such as sending and receiving events and data on ports, or assigning values of variables for the following period.

(2) Communication actions can be freezing the content of incoming ports, initiating a send on an event, data, or event data port, initiating a subprogram call or catching a previously raised execution Timeout exception. Some communication actions include implicit assignments, such as the assignment of actual parameters on subprogram calls (see Z.5.1).[4]

```
basic_action ::=
    skip
    | assignment
    | simultaneous_assignment
    | communication_action
    | timed_action
    | when_throw
    | combinable_operation
    | issue_exception
    | computation_action
```

Z.6.4.1 Skip

(1) A `skip` action does nothing at all.

Semantics

(S1) The weakest precondition of `skip` is the same as its postcondition.

SKIP **[S]**:
$\mathfrak{M}_t[\![\text{wp}(\texttt{skip}, Q)]\!] \equiv \mathfrak{M}_t[\![Q]\!]$ *(skip changes nothing)*

Z.6.4.2 Assignment

(1) An assignment evaluates an expression and binds a variable to that value. When the variable name is followed by a `'`, the value is bound to the variable one period hence.

(2) Assignments consist of a value expression and a target reference for the value assignment separated by the assignment symbol `:=`. When an assignment action is performed, the result of the evaluation of the right hand side expression is stored into the entity specified by the left hand side target reference. Target

[3] BA D.6(2)
[4] BA D.6(4)

references of assignments are local variables, data components acting as persistent state variables, and outgoing features such as ports and parameters.[5]

(3) When assignment actions are used in concurrent composition, then the assigned values are not accessible to expressions of other assignment actions in the same concurrent composition by naming the assignment target.[6]

(4) The keyword `any` should be used to represent non-deterministic behaviors. The purpose of `any` is to represent easily that the assigned value could take any of the possible value determined by the data type. This could be used for formal verification or for simulation purpose using a randomly generated value. The `any` keyword is incompatible with the use of code generation techniques.[7]

```
assignment ::= variable_name [ ' ] := ( expression | record_term | any )
```

(5) When assigning a variable of record type, the value can be expressed as *record term*.

```
record_term ::= ( { record_value }+ )

record_value ::= field_identifier => value ;
```

Consistency Rule

(C1) The type of the assigned value must be consistent with the type of the assignment target. The corresponding literal values are acceptable values for those types.[8]

Legality Rules

(L1) Only periodic components may delay assignment using `'`.

(L2) In an assignment action, the type of the value expression must match the type of the target.[9]

Semantics

(S2) The effect of assigning the value of an expression to a variable is defined using weakest precondition predicate transformers. Where Q is an Assertion, n is a variable name, e is an expression, t is the time of assignment, d is the duration of the period of a periodic component, and $Q|^n_e$ means to replace every occurrence of expression e in Q with variable name n:

THREAD ASSIGNMENT **[TA]**:

$\mathfrak{M}_t[\![\text{wp}(\texttt{n:=e}, Q)]\!] \equiv \mathfrak{M}_t[\![Q|^n_e]\!]$ *(wp by substitution of variable with expression)*

$\mathfrak{M}_t[\![\text{wp}(\texttt{n':=e}, Q)]\!] \equiv \mathfrak{M}_{t+d}[\![Q|^n_e]\!]$ *(time-shifted wp by substitution)*

Z.6.4.3 Simultaneous Assignment

(1) Simultaneous assignment for components is the same as that for subprograms, but allows assignment of next values of variables.

[5] BA D.6(3)
[6] BA D.6(15)
[7] BA D.6(21)
[8] BA D.6(16)
[9] BA D.6(L1)

```
simultaneous_assignment ::=
  ( variable_name [ ' ] { , variable_name [ ' ] }+
  :=
  ( expression | record_term | any )
    { , ( expression | record_term | any ) }+ )
```

Semantics

(S3) Where Q is an Assertion, n_1, n_2, n_3, ..., are variable names, and e_1, e_2, e_3, ..., are expressions, and $Q^{n_1,n_2,n_3,\ldots}_{e_1,e_2,e_3,\ldots}$ means to replace every occurrence of variable name n_x listed with the expression e_x in the corresponding position in Q:

THREAD SIMULTANEOUS ASSIGNMENT **[TSA]**:

$\mathfrak{M}_t[\![\text{wp}(\texttt{(n1,n2,n3,...:= e1,e2,e3,...)},Q)]\!] \equiv \mathfrak{M}_t[\![Q^{n_1,n_2,n_3,\ldots}_{e_1,e_2,e_3,\ldots}]\!]$
(wp by substituting all listed variables with corresponding expression as in §??)
$\mathfrak{M}_t[\![\text{wp}(\texttt{(n1',n2',n3',...:= e1,e2,e3,...)},Q)]\!] \equiv \mathfrak{M}_{t+d}[\![Q^{n_1,n_2,n_3,\ldots}_{e_1,e_2,e_3,\ldots}]\!]$ *(time-shifted substitution of variables by expressions in postcondition)*

Z.6.4.4 Computation Action

(1) A *computation action* models the duration of execution for scheduling, and timing analysis.[10] Presumably implementation will replace communication actions with behavior actions, and derive information for scheduling and timing from simulations or analyses of compiled code.[11]

(2) **computation**`(min .. max)` expresses the use of the CPU for a duration between `min` and `max`. The time is specified in terms of time units as defined by the `Time_Units` property type in the core standard. One value can be specified when `min` and `max` are the same.[12]

```
computation_action ::=
  computation ( behavior_time [ .. behavior_time ] )
  [ in binding ( processor_unique_component_classifier_reference
    { , processor_unique_component_classifier_reference }+ ) ]

behavior_time ::= integer_expression unit_identifier
```

Legality Rule

(L3) The unit identifier must be a time unit.

(L4) The time values must be integers.

(L5) The value of the `max` time must be greater than or equal to the value of the `min` time.[13]

Semantics

[10] **Reconciliation:** computation action
[11] BA D.6(5)
[12] BA D.6(18)
[13] BA D.6(L8)

(S4) When a single behavior-time is used, that defines the difference between suspension and dispatch times.

(S5) When two behavior-times are used, that defines the allowed range in the difference between suspension and dispatch times.

Z.6.4.5 Issue Exception

An *issue exception* action forces transition to an identified state, and send the message string to the implicit `Exception out event data port`.[14]

```
issue_exception ::=
  exception ( [ exception_state_identifier , ] message_string_literal )
```

Z.6.5 Sequential Composition

(1) Sequential composition of actions performs them one after another, in order of appearance.[15]

```
sequential_composition ::=
   asserted_action { ; asserted_action }+
```

Semantics

(S1) Where `S1` and `S2` are formulas, and i, j, and m are intervals:

$\mathfrak{M}_{\mathrm{i}}[\![\texttt{S1;S2}]\!] \equiv \exists \mathrm{j} \subset \mathrm{i}, \mathrm{m} \subset \mathrm{i} \mid \mathfrak{M}_{\mathrm{j}}[\![\texttt{S1}]\!] \wedge \mathfrak{M}_{\mathrm{m}}[\![\texttt{S2}]\!] \wedge start(\mathrm{m}) = end(\mathrm{j})$
(there exist subintervals j *and* m *of* i *such that* j *satisfies* `S1`, m *satisfies* `S2`, *and the least element of* m *is the upper bound of* j*)*

Sequential composition is depicted as sequential lattice combination, $\mathrm{i}_1 \curvearrowright \mathrm{i}_2$, in Figure 1.3.

(S2) Equivalently, `S1 ; S2` has the behavior of an automata transition $T(s,g,d,f)[S1;S2]$ translated to the transition system $T \Rightarrow T_1 \cup T_2$ where $T_1 = T(s,g,e,x)[S1]$ and $T_2 = T(e,x,d,f)[S2]$ by introducing a new execution state e and clock formula x. Sequential composition of more than two actions uses this translation inductively.

Inference Rules

$$\textsc{Sequential Composition: [SC]}\ \frac{\begin{array}{c}\ll P \gg\ S_1\ \ll R_1 \wedge R_2 \gg \\ \ll R_1 \wedge R_2 \gg\ S_2\ \ll Q \gg\end{array}}{\ll P \gg\ S_1\ \ll R_1 \gg\ ;\ \ll R_2 \gg\ S_2\ \ll Q \gg}$$

[14] BLESS Differs from BA: issue exception
[15] BA D.6(11)

SEQUENTIAL COMPOSITION OF K ASSERTED ACTIONS:

$$\textbf{[SCk]}\ \frac{\begin{array}{c}\ll P_1 \gg\ S_1\ \ll Q_1\ \wedge\ P_2 \gg \\ \ll Q_1\ \wedge\ P_2 \gg\ S_2\ \ll Q_2 \wedge P_3 \gg \\ \cdots \\ \ll Q_{k-1}\ \wedge\ P_k \gg\ S_k\ \ll Q_k \gg\end{array}}{\ll P_1 \gg\ S_1\ \ll Q_1 \gg\ ;\ \ll P_2 \gg\ S_2\ \ll Q_2 \gg\ ;\ \cdots\ ;\ \ll P_k \gg\ S_k\ \ll Q_k \gg}$$

Examples

```
la:=StartButton
  <<(la=StartButton) and (Rx_APPROVED())@now  and PB_DURATION()>>
;
Infusion_Flow_Rate!(Basal_Rate)    --infuse at basal rate
  <<(la=StartButton) and (Infusion_Flow_Rate@now=Basal_Rate@now)
    and PB_DURATION()>>
```

```
      <<VS(now) and LAST_AS(now) and LAST_AP(now)>>
    vs!
      <<vs@now and LAST_AS(now) and LAST_AP(now) and AXIOM_CCI()
      and AXIOM_LRLi_gt_URLi_LIMIT(now)>>
    ;
    cci!(now-last_vp_or_vs)
      <<vs@now and LAST_AS(now) and LAST_AP(now)
         and AXIOM_LRLi_gt_URLi_LIMIT(now)>>
    ;
    last_vp_or_vs := now
      <<(last_vp_or_vs=now) and vs@now and LAST_AS(now) and LAST_AP(now)
        and AXIOM_LRLi_gt_URLi_LIMIT(now)>>
```

```
<<E() and ACTUAL_POSITION>0 and ACTUAL_IN_RANGE()>>
  Delta := -1                          --set the delta
  <<E() and Delta= -1 and (ACTUAL_POSITION-1)>=0 and AXIOM_GT(ACTUAL_POSITION)
      and ACTUAL_POSITION<=PCS::MaxPosition>>
  ;  --close valve one step
   ActuatorCommand(pc:Delta)
  <<ACTUAL_POSITION'=(ACTUAL_POSITION+Delta) and Delta= -1
      and (ACTUAL_POSITION-1)>=0 and E()
      and (ACTUAL_POSITION-1)<=PCS::MaxPosition>>
  ;  --set own estimate of position
  EstimatedActualPosition' := (EstimatedActualPosition-1)
  <<EstimatedActualPosition'=ACTUAL_POSITION'
    and ACTUAL_POSITION'>=0 and ACTUAL_POSITION'<=PCS::MaxPosition>>
```

Z.6.6 Concurrent Composition

(1) Concurrently-composed actions are order independent; the actions may be performed in any order, or concurrently with the same result.[16]

```
concurrent_composition ::=
   asserted_action { & asserted_action }+
```

Legality Rules

(L1) The same local variable must not be assigned in different actions of a concurrent composition.[17]

(L2) The same port must not be assigned in different actions of a concurrent composition.[18]

Semantics

(S1) Where S_1 and S_2 are actions; P and Q are assertions:

CONCURRENT COMPOSITION: **[CC]**
$$\frac{P \rightarrow \text{wp}(S_1, Q) \quad P \rightarrow \text{wp}(S_2, Q)}{\ll P \gg\ S_1\ \&\ S_2\ \ll Q \gg}$$

(S2) Where $A_1, A_2, \ldots, A_k$ are asserted actions: $A_j = \ll P_j \gg\ S_j\ \ll Q_j \gg$ for $j \in 1..k$; P and Q are assertions:

CONCURRENT COMPOSITION OF K ASSERTED ACTIONS:

[CCk]
$$\frac{\begin{array}{c} P \rightarrow P_1, P \rightarrow P_2, \cdots, P \rightarrow P_k \\ P_1 \rightarrow \text{wp}(S_1, Q_1) \\ P_2 \rightarrow \text{wp}(S_2, Q_2) \\ \cdots \\ P_k \rightarrow \text{wp}(S_k, Q_k) \\ Q_1 \wedge Q_2 \wedge \cdots \wedge Q_k \rightarrow Q \end{array}}{\ll P \gg\ \{A_1\ \&\ A_2\ \&\ \ldots\ \&\ A_k\}\ \ll Q \gg}$$

In general, when the optional precondition P_j is omitted from asserted action A_j, then P may be used in its place. When all of the optional postconditions Q_j are omitted, then Q may be used for each. If any postconditions Q_j are included, then **true** may be used for omitted postconditions.

(S3) Concurrent composition is depicted as concurrent lattice combination, $\underline{i}_1 \Downarrow \underline{i}_2$, in Figure 1.3. Where S_1 and S_2 are actions, and i, j, and m are intervals:

[16] BA D.6(11)
[17] BA D.6(L3)
[18] BA D.6(L4)

$$\mathfrak{M}_{\underline{i}}[\![S_1]\!] \wedge \mathfrak{M}_{\underline{m}}[\![S_2]\!],$$
$$\mathfrak{M}_{\underline{i}}[\![\texttt{S1\&S2}]\!] \equiv \exists \underline{j} \subset \underline{i}, \underline{m} \subset \underline{i} \mid \; start(\underline{i}) = start(\underline{m}) = start(\underline{j}),$$
$$end(\underline{i}) = end(\underline{m}) = end(\underline{j})$$

(there exist subintervals $\underline{j}$ *and* $\underline{m}$ *of* $\underline{i}$ *such that* $\underline{j}$ *satisfies* `S1`, $\underline{m}$ *satisfies* `S2`, *and* $\underline{i}$, $\underline{j}$, *and* $\underline{m}$ *share least elements and upper bounds)*

Semantics for more than two concurrently-composed actions are defined inductively.

(S4) Equivalently, `S1 &S2` has the behavior of an automata transition $T(s,g,d,f)[S1\&S2]$ translated to the synchronous composition[19] $(T_1|T_2)[(s,s)/s,(d,d)/d]$ of transition systems where $T_1 = T(s,g,d)[S1]$ and $T_2 = T(s,g,d)[S2]$ substituting the composed states (s,s) and (d,d) by s and d.

Example

```
T18_VRP_EXPIRED :  --vs after VRP expired
check_vrp -[sv? and not tnv? and (vrp<=(now-last_vp_or_vs))]-> va
  {<<VS(now) and LAST_VP_OR_VS(now) and LAST_AS(now) and LAST_AP(now)>>
  vs!
   <<vs@now and LAST_AS(now) and LAST_AP(now)>>
      -- and AXIOM_LRLi_gt_URLi_LIMIT(now)
  &
  cci!(now-last_vp_or_vs)
  &
  last_vp_or_vs := now
   <<(last_vp_or_vs=now) and LAST_VP_OR_VS(now)>>};
```

Z.6.7 Alternative

(1) An *alternative* action using guarded actions (or commands) makes the proof semantics symmetric. A boolean expression *guards* each alternative; guards may be evaluated in any order.[20] At least one of the guards must be true. If more than one guard is true, any of their alternatives may be performed.

(2) An alternative action using if-elseif-else makes semantics asymmetric.[21] The order is now significant in that cascading alternatives assume that no previous alternative was taken. Sometimes, that important, sometimes not, which leads to misunderstanding and error.

```
alternative ::=
   if  guarded_action { [] guarded_action }+ fi
   |
   if ( boolean_expression_or_relation ) behavior_actions
   { elsif ( boolean_expression_or_relation ) behavior_actions }*
   [ else behavior_actions ]
   end if
guarded_action ::=
   ( boolean_expression_or_relation )~> behavior_actions
```

[19] put reference to synchronous composition here
[20] BLESS Differs from BA: if [] fi
[21] **Reconciliation:** add if-elsif-else

Legality Rules

(L1) At least one of the guards must be true.

(L2) The weakest precondition of alternative is least one guard must be true.

Semantics

(S1) The semantics of if-fi alternative is classic[22] guarded commands.

$$\mathfrak{M}_{\underline{i}}[\![\texttt{if (B1)->S1[](B2)->S2[]}\cdots\texttt{[](Bn)->Sn fi}]\!] \equiv \begin{array}{r} \mathfrak{M}_{start(\underline{i})}[\![B_1]\!] \rightarrow \mathfrak{M}_{\underline{i}}[\![S_1]\!], \\ \mathfrak{M}_{start(\underline{i})}[\![B_2]\!] \rightarrow \mathfrak{M}_{\underline{i}}[\![S_2]\!], \\ \vdots \\ \mathfrak{M}_{start(\underline{i})}[\![B_n]\!] \rightarrow \mathfrak{M}_{\underline{i}}[\![S_n]\!], \\ \mathfrak{M}_{start(\underline{i})}[\![B_1]\!] \vee \mathfrak{M}_{start(\underline{i})}[\![B_2]\!] \vee \cdots \vee \mathfrak{M}_{start(\underline{i})}[\![B_n]\!] \end{array}$$

(whenever a guard is true at the beginning of interval i, its action will be true over all of i, and at least one of the guards is true)

(S2) Equivalently, `if (B1)->S1[](B2)->S2[]fi` has the behavior of an automata transition $T(s,g,d,f)$[`if (B1)->S1[](B2)->S2[]fi`] translated to the union of transition systems $T \Rightarrow T_1 \cup T_2$ where $T_1 = T(s, g \wedge B1, d)[S1]$ and $T_2 = T(s, g \wedge B2, d)[S2]$. An arbitrary number of alternatives is defined similarly making a transition system for each alternative. At least one alternative guard must be true.

(S3) The semantics of if-elsif-else alternative is defined in terms of and equivalent if-fi alternative.

$$\mathfrak{M}_{\underline{i}}[\![\texttt{if (B1) S1 elsif (B2) S2 ... elsif (Bn) Sn else Sm end if}]\!]$$

$$\equiv \begin{array}{r} \mathfrak{M}_{start(\underline{i})}[\![B_1]\!] \rightarrow \mathfrak{M}_{\underline{i}}[\![S_1]\!], \\ \mathfrak{M}_{start(\underline{i})}[\![B_2]\!] \wedge \neg\mathfrak{M}_{start(\underline{i})}[\![B_1]\!] \rightarrow \mathfrak{M}_{\underline{i}}[\![S_2]\!], \\ \vdots \\ \mathfrak{M}_{start(\underline{i})}[\![B_n]\!] \wedge \neg\mathfrak{M}_{start(\underline{i})}[\![B_1]\!] \ldots \wedge \neg\mathfrak{M}_{start(\underline{i})}[\![B_{n-1}]\!] \rightarrow \mathfrak{M}_{\underline{i}}[\![S_n]\!], \\ \neg\mathfrak{M}_{start(\underline{i})}[\![B_1]\!] \ldots \wedge \neg\mathfrak{M}_{start(\underline{i})}[\![B_n]\!] \rightarrow \mathfrak{M}_{\underline{i}}[\![S_m]\!] \end{array}$$

(S4) Equivalently, `if (B1) S1 elsif (B2) S2 else Sm end if` has the behavior of an automata transition $T(s,g,d,f)$[`if (B1) S1 elsif (B2) S2 else Sm end if`] translated to the union of transition systems $T \Rightarrow T_1 \cup T_2 \cup T_m$ where $T_1 = T(s, g \wedge B1, d)[S1]$, $T_2 = T(s, g \wedge B2 \wedge \neg B1, d)[S2]$, and $T_m = T(s, g \wedge \neg B2 \wedge \neg B1, d)[Sm]$. An arbitrary number of alternatives is defined similarly making a transition system for each alternative.

Inference Rules

(S5) Where `B1` , `B2`, and `Bn` are boolean-valued expressions, and `S1`, `S2`, and `Sn` are actions:[23]

[22] Dijkstra-Gries

[23] The $\cdots$ represent elided guarded actions.

ALTERNATIVE **[IF]**: $\text{wp}(\texttt{if (B1)->S1[](B2)->S2[]}\cdots\texttt{[](Bn)->Sn fi}, Q) \equiv$

$$\begin{array}{l} B_1 \vee B_2 \vee \cdots \vee B_n, \\ (B_1 \rightarrow \text{wp}(S_1, Q)), \\ (B_2 \rightarrow \text{wp}(S_2, Q)) \\ \vdots \\ (B_2 \rightarrow \text{wp}(S_2, Q)) \end{array}$$

Examples

```
if
  --good SpO2 reading, reset counter
  (SensorConnected? and not MotionArtifact?)~>
    <<SensorConnected^0 and not MotionArtifact^0>>
  numBadReadings := 0
    <<NUMBAD()>>
[]  --bad SpO2, not enough bad reading to alarm
  (MotionArtifact? or not SensorConnected?)~>
    <<all j:integer in 0 ..numBadReadings are
       MotionArtifact^(-j) or not SensorConnected^(-j)>>
  numBadReadings :=numBadReadings+1
    <<NUMBAD()>>
fi
```

```
if (SensorConnected? and not MotionArtifact?)
then
  numBadReadings := 0
else
  numBadReadings :=numBadReadings+1
end if
```

```
if
  (guard_A)~> action_A
[]
  (guard_B)~> action_B
[]
  (guard_C)~> action_C
[]
  (guard_D)~> action_D
fi
```

Z.6.8 Behavior Action Block

(1) A *behavior action block* (optionally) introduces local variables of bounded type and lifetimes.

```
behavior_action_block ::=
   [ quantified_variables ] { behavior_actions }
   [ timeout behavior_time ]  [ catch_clause ]
```

(2) The optional `catch_clause` allows specification of behavior upon occurrence of exceptions as defined in Z.6.11, Exception Handling.[24]

(3) Quantified variables are local variables, and exist only during lattice construction. Behavior variables are defined in Z.3.3, Behavior Variables.

```
quantified_variables ::= declare { behavior_variable }+
```

Legality Rule

(L1) Timeout on behavior actions are not allowed on behavior transitions with timeout conditions.[25]

Semantics

(S1) Where v is a variable identifier,t is a type, e is an expression, and S is a formula:

$\mathfrak{M}_i[\![$ `<<P>> declare v:t:=e; {S} <<Q>>` $]\!]$

$\equiv$ $\exists$ v∈t | $\mathfrak{M}_{start(i)}[\![v]\!] = \mathfrak{M}_{start(i)}[\![e]\!]$
$\wedge\ \mathfrak{M}_i[\![S]\!]\ \wedge\ \mathfrak{M}_{start(i)}[\![P]\!] \wedge \mathfrak{M}_{end(i)}[\![Q]\!]$

(there exists a variable v of type t, where v equals the value of e evaluated at at start(i), P is true at start(i), Q is true at end(i), and i satisfies S)

$\mathfrak{M}_i[\![$ `{S}` $]\!] \equiv \mathfrak{M}_i[\![S]\!]$

(the meaning of braces without quantified variables is its contents)

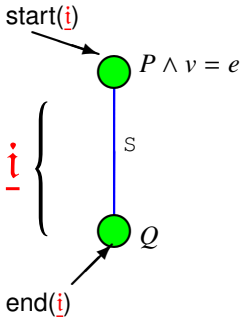

Figure Z.6.1: Block

Inference Rule

$$\textsc{Block:}\ [\mathbf{B}]\ \frac{\begin{array}{c}\exists v \in t \mid \ll P \wedge v = e \gg \ \rightarrow\ \ll A \gg \\ \ll A \gg\ S\ \ll B \gg \\ \ll B \gg \rightarrow \ll Q \gg\end{array}}{\ll P \gg\ \texttt{declare v:t:=e; \{<<A>>S <<B>>\}}\ \ll Q \gg}$$

(S2) Equivalently, `<<P>> declare v:t:=e; {<<A>> S <<B>>} <<Q>>` has the behavior of an automata transition $T(s, v = e, d, true)[S]$ from state s in which assertion $\ll P \gg$ holds, to state d in which assertion $\ll Q \gg$ holds while performing action S. Additionally, assertion $\ll A \gg$ must be derivable from $\ll P \gg$ with the initial value of v, $\ll P \wedge v = e \gg \rightarrow \ll A \gg$, and also $\ll B \gg \rightarrow \ll Q \gg$ in which v may appear in B, but not Q.

Example

Block from cardiac pacemaker rate controller:

```
declare  --transient, local variables
  siri : real := (msr>(lrl-(f*(xl-thresh)))??msr : lrl-(f*(xl-thresh)));
```

[24] BLESS Differs from BA: catch clause
[25] BA D.3(L11)

```
    z : real := ((lrl-msr)*(lrl+msr)) / (2*(rt-lrl));
    y : real := ((lrl-msr)*(lrl+msr)) / (2*(ct-lrl));
    up_siri : real := ((cci-z)<siri ?? siri : cci-z);
    dn_siri : real := ((cci+y)<siri ?? cci+y : siri);
    down : real :=  cci*(1.0+(drs/100.0));  --down rate smoothing
    up : real := cci*(1.0-(urs/100.0));  --up rate smoothing
  {
  <<((lrl-url)<>0) and (z=Z()) and (y=Y())
      and (siri=SIRi()) and (dn_siri=DN_SIRi()) and (up_siri=UP_SIRi())
      and (down=DOWN()) and (up=UP())>>
  dav!((cci*((av-min_av)/(lrl-url))) + min_av)
  &
  min_cci!((url>(up_siri > up??up_siri: up)??url:(up_siri >up??up_siri:up)))
  &
  max_cci!((lrl<(dn_siri<down??dn_siri:down)??lrl:(dn_siri<down??dn_siri:down)))
  <<true>>
  }
```

Z.6.9 Forall

(1) To specify concurrent execution of many similar actions *forall action* defines local variables restricted to a range, that may then be used as variables within its block

```
forall_action ::=
  forall variable_identifier { , variable_identifier }*
    in integer_expression .. integer_expression  behavior_action_block
```

Semantics

(S1) Two, identical semantics for forall action are given: weakest-precondition predicate transformer and inference rule. Weakest-precondition is much preferred, but can only be used when the wp of the body is known. Semantics for multiple quantified variables is the same as replacing "a" with a sequence of variable identifiers.

(S2) The weakest-precondition predicate transformer for forall action, is the conjunction of the weakest precondition predicate transformed bodies with the quantified variable replaced by each value in the range, and that those transformed, substituted bodies are interference free.[26] The body that uses the quantified variable is $S(a)$.

FORALL ACTION **[FA]**: wp(`forall a in R {S(a)}`, Q) $\equiv$ $\forall a \in R \,|\, \text{wp}(S(a), Q)$, $\forall a \in R \,|\, \text{interference-free}(S(a))$

[26] Interference freedom is none of the concurrent actions assigns values that other actions either use or assigns.

$$\mathfrak{M}_{\underline{i}}[\![\texttt{forall a in R \{<<p(a)>>S(a)<<q(a)>>\}}]\!] \equiv \begin{array}{l} \exists \underline{i}_1 \in \underline{i}, \underline{i}_2 \in \underline{i}, \ldots, \underline{i}_n \in \underline{i}, | \\ \underline{i} = \underline{i}_1 \Downarrow \underline{i}_2 \Downarrow \ldots \Downarrow \underline{i}_n \\ \forall a \in R \mid \mathfrak{M}_{start(\underline{i})}[\![p_a]\!] \\ \forall a \in R \mid \mathfrak{M}_{\underline{i}}[\![\texttt{<<p(a)>>S(a)<<q(a)>>}]\!] \\ \forall a \in R \mid \mathfrak{M}_{end(\underline{i})}[\![q_a]\!] \end{array}$$

(ya do the hokie pokie and you turn yourself around)

Inference Rule

(S3) Where $B(i)$ is the value of B after the ith iteration, E is a boolean-valued expression, B is an integer-valued function, S is an action, and P, I, and Q are assertions:

FORALL ACTION: **[FA]**

$$\frac{\begin{array}{c} P \rightarrow \forall a \in [lb..ub]\ R_a \\ \ll R_a \gg\ S_a\ \ll T_a \gg \\ \forall a \in [lb..ub]\ T_a \ \rightarrow\ Q \\ \forall a \neq b \in [lb..ub]\ \text{interference-free}(S_a, S_b) \end{array}}{\ll P \gg \texttt{ forall a:t in lb..ub \{<<R>>S<<T>>\}} \ll Q \gg}$$

(to prove forall action requires: the precondition imply all inner preconditions, the body is correct, all inner postconditions together imply the postcondition, and all actions are inteference-free)

(S4) Equivalently, `forall a:t in lb..ub <<R>>S<<T>>` has the behavior of an automata transition $T(s,g,d)$[`forall a:t in lb..ub S`] translated to the union of transition systems $T \Rightarrow T_1 \cup T_2 \ldots \cup T_m$ where $T_1 = T(s, g \wedge (t = lb), d)[S]$, $T_2 = T(s, g \wedge (t = lb + 1), d)[S]$, and $T_m = T(s, g \wedge (t = ub), d)[S]$. An arbitrary number of alternatives is defined similarly making a transition system for each alternative.

Example

```
   <<SpO2_INV() and (num_samples<PulseOx_Properties::Num_Trending_Samples)>>
 forall i:integer in 1 ..num_samples
   {
     <<spo2[i]=(MotionArtifact^(-i) or not SensorConnected^(-i)??0:SpO2^(-i))>>
    spo2_nxt[i+1]:=spo2[i] --shift old samples
      <<spo2_nxt[i+1]=(MotionArtifact^(-i) or not SensorConnected^(-i)
           ??0:SpO2^(-i))>>
    }
   <<SHFT: :all i:integer in 1 ..num_samples
    are spo2_nxt[i+1]=(MotionArtifact^(-i) or not SensorConnected^(-i)
         ??0:SpO2^(-i))>>
```

Z.6.10 Loops

(1) Loops allow actions to be repeated in some controlled manner.

Z.6.10.1 While Loop

(1) The *while loop* repeats an action while a guard (boolean expression) is true. While loops may have invariant assertion, and a bound function. The *invariant* must be true before and after each iteration. The *bound function* when positive must imply the guard is true; the bound function when zero or less must imply the guard is false; and, each iteration of the loop must decrease the value of the bound function.

```
while_loop ::=
   while ( boolean_expression_or_relation )
   [ invariant assertion ]
   [ bound integer_expression ]
   behavior_action_block
```

Semantics

(S1) Where $B(i)$ is the value of B after the ith iteration, E is a boolean-valued expression, B is an integer-valued function, S is an action, and P, I, and Q are assertions:

$$\text{Loop: } \textbf{[L]} \frac{\begin{array}{c} P \rightarrow I \\ I \rightarrow \text{wp}(S, I) \\ (I \wedge \neg E) \rightarrow Q \\ B > 0 \rightarrow E \\ B(i) > B(i+1) \end{array}}{\ll P \gg \texttt{ while (E) invariant <<I>> bound B \{S\} } \ll Q \gg}$$

(S2) Equivalently, `while (E) S` has the behavior of an automata transition $T(s, g, d, f)$[`while (E) S`] translated to the union of transition systems $T \Rightarrow T_1 \cup T_2 \cup T_3 \cup T_4$ where $T_1 = T(s, g \wedge E, c)[S]$, $T_2 = T(c, E, c)[S]$, $T_3 = T(c, \neg E, d, f)[S]$, and $T_4 = T(s, \neg E, d, f)$, by introducing a new execution state c. If defined, invariant `<<I>>` must hold for states s, d, and c.

Example

```
        while ((dc <> dl[k].code) and ((PCA_Properties::Drug_Library_Size-k)>0))
          invariant <<INVW()>>
          bound (PCA_Properties::Drug_Library_Size-k)
          {
          <<((PCA_Properties::Drug_Library_Size-k)>0) and INVW()>>
          k:=k+1
          <<(0<(PCA_Properties::Drug_Library_Size-(k-1))) and INVW()>>
          }
        <<INVW() and not
           ((dc<>dl[k].code) and ((PCA_Properties::Drug_Library_Size-k)>0))>>
```

Z.6.10.2 For Loop

(1) A *for loop* is a handy specialization of a while loop, that introduces an integer variable, defined over an integer range, implicitly initialized at the lower bound, incremented after each iteration, and loop termination after the variable equals the upper bound.

```
for_loop ::=
   for integer_identifier
      in integer_expression .. integer_expression
   [ invariant assertion ]
   { asserted_action }
```

Naming Rule

(N1) The integer identifier of a for control construct represents a variable whose scope is local to the for construct. Such a variable must not be otherwise be visible in scope.[27]

Legality Rules

(L1) The lower bound must be at most the upper bound.

(L2) An integer identifier of a for loop is not a valid target for an assignment action.[28]

Semantics

(S3) Where `a` is a fresh integer variable, `lb` and `ub` are integer-valued expressions for the lower-bound and upper-bound respectively, `I` is a predicate invariant before and after each execution of the loop, and `S(a)` are behavior actions that use `a`:

$$\textsc{For}:\ \textbf{[FOR]}\quad \frac{\begin{array}{c}\texttt{lb}\leq\texttt{ub},\\ \ll P\gg\\ \texttt{variables a:Ideal::integer:=lb;}\\ \texttt{\{while (a<=ub) invariant <<I>> bound ub-a \{S(a);a:=a+1\}\}}\\ \ll Q\gg\end{array}}{\ll P\gg\ \texttt{for (a in lb..ub) invariant <<I>> \{S(a)\}}\ \ll Q\gg}$$

(S4) Equivalently, `for (a in lb..ub) S(a)` has the behavior of an automata transition $T(s,g,d)$[`for (a in lb..ub) S(a)`] translated to the union of transition systems $T \Rightarrow T_1 \cup T_2 \ldots \cup T_m$ where $T_1 = T(s,g,c_1)[S(lb)]$, $T_2 = T(c_1,true,c_2)[S(lb+1)]$, and $T_m = T(c_{m-1},true,d)[S(ub)]$, by introducing a new execution states $c_1 \ldots c_{m-1}$, where $m = (ub - lb) + 1$. If defined, invariant `<<I>>` must hold for states s, d, and $c_1 \ldots c_{m-1}$.

Example

```
for (i in lb..ub)
  invariant <<A()>>
  {
  h[i] := g[i]
```

[27] BA D.6(N1)
[28] BA D.6(L2)

```
    }
```

Z.6.10.3 Do-Until Loop

(1) A *do-until* loop is another specialization of a while loop in which the body is executed unconditionally before evaluating the guard.

```
do_until_loop ::=
    do
    [ invariant assertion ]
    [ bound integer_expression ]
    behavior_actions
    until ( boolean_expression_or_relation )
```

Semantics

(S5) Where $B(i)$ is the value of B after the ith iteration, E is a boolean-valued expression, B is an integer-valued function, S is behavior actions, and P, I, and Q are assertions:

DO-UNTIL: **[UNTIL]**

$$\frac{\ll P \gg \texttt{ S <<I>>; while (not E) invariant <<I>> bound B \{S\} } \ll Q \gg}{\ll P \gg \texttt{ do invariant <<I>> bound B S until (E) } \ll Q \gg}$$

(S6) Equivalently, `do S until (E)` has the behavior of an automata transition $T(s,g,d,f)$[`do S until (E)`] translated to the union of transition systems $T \Rightarrow T_1 \cup T_2 \cup T_3$ where $T_1 = T(s,g,c)[S]$, $T_2 = T(c, g \wedge \neg E, c)[S]$, and $T_3 = T(c,E,d,f)$, by introducing a new execution state c. If defined, invariant `<<I>>` must hold for states s, d, and c.

Z.6.11 Exception Handling

(1) Safety-critical systems need to define system behavior in every exceptional circumstance. Therefore a way to specify how those exceptions shall be detected, reported, and resolved. BLESS concerns mostly the reporting part plus some detection. Most importantly, BLESS behavior does not resolve exceptions; it just emits an event out a port with an error code. The Error Model Annex[29] (EMV2) was made to be used to define system response to faults like BLESS exceptions.

Grammatically, to catch an exception involves adding an optional catch clause to block. Testing for anomalous conditions and raising exceptions adds another basic action.

```
catch_clause ::= catch { ( exception_label : basic_action ) }+

exception_label ::= { exception_identifier }+ | all
```

[29] SAE International Standard AS5506B Annex E

(2) Using `all` as the exception label will catch every exception with the preceding block. Multiple exceptions may cause the same action, `catch(x1 x2 x3:a)`, or different actions, `catch(x1:a1) (x2 x3:a2)`.

(3) When exceptions are caught by threads, transition to a special state may be forced with the `issue_exception` action (Z.6.4.5). This is not allowed within subprograms, because they don't have states.

(4) Exceptions may be thrown automatically (i.e. divide by zero) or deliberately with a when-throw action.

`when_throw ::= when ( boolean_expression ) throw exception_identifier`

Semantics

(S1) Where v is a variable identifier,t is a type, e is an expression, S is a formula, x is an exception identifier, k is an integer-valued expression, and r!(k) is a basic action that sends an event out error data port r with error code k:

$\mathfrak{M}_i[\![$`declare v:t:=e {S}catch(x:r!(k))`$]\!] \equiv \mathfrak{M}_i[\![$`declare v:t:=e {S}`$]\!] \vee (x \in i \wedge \mathfrak{M}_i[\![$`r!(k)`$]\!])$
(either the lattice is constructed normally, or exception x occurred and value k sent out error port r)

(S2) Semantics for multiple exception labels and actions extends that above.[30]

Example

The following example performs behavior actions when in state `s` and condition `c` is true before transitioning to state `d`. The behavior actions are to do some `work` followed by `morework` concurrently-composed with a when-throw action that raises exception `x`, that when caught sends an event out of port `er`.

```
s -[c]-> d
  {work;{morework&when(badthing)throw x}catch(x:er!)};
```

Z.6.12 Locking Actions

Locking actions are part of the BLESS grammar to retain backward compatibility with BA programs that use them.[31]

The four locking actions:

`*!<` enter critical section

`*!>` leave critical section

`!<` lock data component

`!>` unlock data component

[30] SOMEBODY OUGHT TO WRITE A STANDARD LIST OF BUILT-IN EXCEPTIONS LIKE DIVIDE BY ZERO OR ARITHMETIC OVERFLOW.

[31] **Reconciliation:** locking actions

```
locking_action ::= *!< | *!> |
  required_data_access_name !< |  required_data_access_name !>
```

Locking actions were originally omitted from BLESS because they void the assumption that the time between dispatch and suspension is 'negligible'. Although the definition of 'negligible' has been deliberately left fuzzy, but stopping execution when some other thread had locked a shared data component, or not exited their mutual critical section. is certainly not negligible.

Anyway, locking actions are a rather blunt mechanism to enforce interference freedom. There are much more adroit means in safety-critical embedded system to share information that don't require locking actions.

Legality Rule

(L1) Accesses to shared data components must be used in a way that no complete state can be reached if a resource has been locked (using for instance `Get_Resource`, or `!<`) and not released (using for instance `Release_Resource`, or `!>`).[32]

Semantics

(S1) Data accesses are similarly subject to a communication protocol between the calling behavior (the client) and the parent component owning the data (the server).

- `required_data_access_name !<`
- `required_data_access_name !>`
- `*!< and *!>`

A shared data access lock `dataname!<` is hence encoded by $T(g,s,d)[dataname!<] = (s,g,sds,c),(c,sdf,true,d)$. The output port sds encodes the request to dataname and the input port spf is dispatched when access to dataname is granted. `Get_Resource` and `Release_Resource` actions are treated similarly.

Z.6.13 Combinable Operations

Combinable operations are both indivisible and possibly simultaneous. They allow concurrent access to shared data structures. Crucially, combinable operations upon the same target, have the same effect whether executed individually or simultaneously. All combinable operations have three parameters: a target variable of appropriate type, declared to be `shared`; a value to be used in the operation; and an identifier of a local variable to hold the result. Combinable operations on `shared` variables provide concurrent, interference-free access to `spread` data structures, particularly arrays. Used properly, a set of combinable operations has the same effect executed in any order, or simultaneously.

[32] BA D.6(L7)

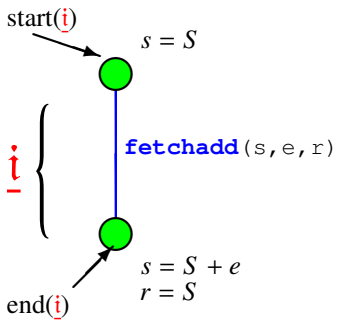

Figure Z.6.2: Single Fetch-Add

```
combinable_operation ::=
  fetchadd
  ( target_variable_name , arithmetic_expression [, result_identifier] )
  |
  ( fetchor | fetchand | fetchxor )
  ( target_variable_name , boolean_expression [, result_identifier] )
  |
  swap
  ( target_variable_name , reference_variable_name , result_identifier )
```

Z.6.13.1 Fetch-Add

(1) A single fetch-add operation has the effect of placing the target variable's value into the result variable while indivisibly incrementing the value of the target variable by the value of the expression. Where s is a shared integer name, e is an integer-valued expression, and r is an identifier of an integer variable

$$\mathfrak{M}_{i}[\![\texttt{fetchadd(s,e,r)}]\!] \equiv \begin{array}{l} \mathfrak{M}_{end(i)}[\![s]\!] = \mathfrak{M}_{start(i)}[\![s]\!] + \mathfrak{M}_{start(i)}[\![e]\!] \\ \mathfrak{M}_{end(i)}[\![r]\!] = \mathfrak{M}_{start(i)}[\![s]\!] \end{array}$$

(the meaning of fetch-add over an interval i, is the meaning of r at the end of i equals s at the start of i, an s at the end of i equals the sum of s and e at the start of i)

(2) When two fetch-add operations target the same shared integer, the result is non-deterministic, however it must be equivalent to *some* series of fetch-adds. Where s is a shared integer name, e_1 and e_2 are integer-valued expressions, r_1 and r_2 are identifiers of integer variables, and F is the text

$$\text{“}\texttt{fetchadd(s,e1,r1) \& fetchadd(s,e2,r2)}\text{”}\,\mathfrak{M}_{i}[\![F]\!] \equiv \begin{array}{l} \mathfrak{M}_{end(i)}[\![s]\!] = \mathfrak{M}_{start(i)}[\![s]\!] + \mathfrak{M}_{start(i)}[\![e_1]\!] + \mathfrak{M}_{start(i)}[\![e_2]\!] \\ \mathfrak{M}_{end(i)}[\![r_1]\!] = \mathfrak{M}_{start(i)}[\![s]\!] \\ \mathfrak{M}_{end(i)}[\![r_2]\!] = \mathfrak{M}_{start(i)}[\![s]\!] + \mathfrak{M}_{start(i)}[\![e_1]\!] \\ \text{OR} \\ \mathfrak{M}_{end(i)}[\![s]\!] = \mathfrak{M}_{start(i)}[\![s]\!] + \mathfrak{M}_{start(i)}[\![e_1]\!] + \mathfrak{M}_{start(i)}[\![e_2]\!] \\ \mathfrak{M}_{end(i)}[\![r_1]\!] = \mathfrak{M}_{start(i)}[\![s]\!] + \mathfrak{M}_{start(i)}[\![e_2]\!] \\ \mathfrak{M}_{end(i)}[\![r_2]\!] = \mathfrak{M}_{start(i)}[\![s]\!] \end{array}$$

(the meaning of concurrent fetch-adds over an interval i, is the meaning of s at the end of i equals the sum of s, e_1, and e_2 at the start of i, and either r_1 at the end of i equals s at the start of i, and r_2 at the

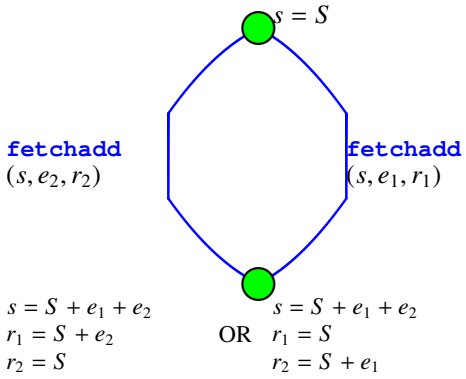

Figure Z.6.3: Two Fetch-Adds

end of i *equals the sum of* s *and* e_1 *at the start of* i*, or* r_2 *at the end of* i *equals* s *at the start of* i*, and* r_1 *at the end of* i *equals the sum of* s *and* e_2 *at the start of* i*)*

(3) If fetch-adds are executed in index order, 1 to n, then the target s will be incremented by the sum of the expressions, each result r_j is the sum of the target and all expressions e_1 to $e_j - 1$, and M is the text "`fetchadd(s,e1,r1) ; ... ; fetchadd(s,en,rn)`"[33] $\mathfrak{M}_i[\![M]\!] \equiv$

$$\begin{array}{l} \mathfrak{M}_{\text{end}(i)}[\![s]\!] = \mathfrak{M}_{\text{start}(i)}[\![s]\!] + \sum_{k=1}^{n} \mathfrak{M}_{\text{start}(i)}[\![e_k]\!] \\ \mathfrak{M}_{\text{end}(i)}[\![r_1]\!] = \mathfrak{M}_{\text{start}(i)}[\![s]\!] \\ \forall j \in 2..n \mid \mathfrak{M}_{\text{end}(i)}[\![r_j]\!] = \begin{array}{l} \mathfrak{M}_{\text{start}(i)}[\![s]\!] \\ + \sum_{k=1}^{j-1} \mathfrak{M}_{\text{start}(i)}[\![e_k]\!] \end{array} \end{array}$$

(4) In the general case of n concurrent fetch-adds, requires use of non-deterministic permutations from §1.5. For that, a sequence P is defined to be a permutation of the numbers 1 to n, to indicate any ordering of n fetch-adds, with $P(j)$ being the jth element in P, and $P^{-1}(j)$ being the index of the element of P that holds j. Let C be the text "`fetchadd(s,e1,r1) & ... & fetchadd(s,en,rn)`"

$$\begin{array}{l} \mathfrak{M}_{\text{end}(i)}[\![s]\!] = \mathfrak{M}_{\text{start}(i)}[\![s]\!] + \sum_{k=1}^{n} \mathfrak{M}_{\text{start}(i)}[\![e_k]\!] \\ \exists P \leftrightarrows (1, \ldots, n) \mid \forall j \in 1..n \mid \\ \mathfrak{M}_{\text{end}(i)}[\![r_j]\!] = \begin{array}{l} \mathfrak{M}_{\text{start}(i)}[\![s]\!] + \\ \sum_{k=1}^{P^{-1}(j)-1} \mathfrak{M}_{\text{start}(i)}[\![e_{P^{-1}(k)}]\!] \end{array} \end{array}$$

$\mathfrak{M}_i[\![C]\!] \equiv$

(the target is incremented by sum of the fetch-add parameters; and the results if the fetch-adds occurred in a arbitrary order)

(5) Sometimes, the return value of fetch-add is not needed and omitted. Let C_2 be the text "`fetchadd(s,e1) & ... & fetchadd(s,en)`"

$\mathfrak{M}_i[\![C_2]\!] \equiv \mathfrak{M}_{\text{end}(i)}[\![s]\!] = \mathfrak{M}_{\text{start}(i)}[\![s]\!] + \sum_{k=1}^{n} \mathfrak{M}_{\text{start}(i)}[\![e_k]\!]$

(the target is incremented by sum of the fetch-add parameters)

(6) However, usually the parameter value is constant 1 or -1. Let I be the text "`fetchadd(s,1,r1) & ... & fetchadd(s,1,rn)`"

[33] semicolon separates elements of action sequences

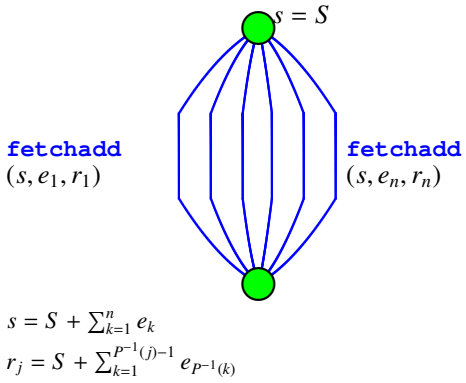

Figure Z.6.4: Many Concurrent Fetch-Adds

$$\mathfrak{M}_{\underline{i}}[\![I]\!] \equiv \begin{array}{l} \mathfrak{M}_{\text{end}(i)}[\![s]\!] = \mathfrak{M}_{\text{start}(\underline{i})}[\![s]\!] + n \\ \exists P \leftrightarrows (1,\ldots,n) \mid \forall j \in 1..n \mid \\ \quad \mathfrak{M}_{\text{end}(i)}[\![r_j]\!] = \mathfrak{M}_{\text{start}(\underline{i})}[\![s]\!] + P^{-1}(j) - 1 \end{array}$$

(the target is incremented by the number of fetch-add-one operations $s + n$; and the results are some non-deterministic permutation of $(s, \ldots, s + n - 1)$)

That each of the results are both in range and different, will be used to for concurrently-accessible data structures to assure interference-freedom. This is the classic "Deli" algorithm wherein patrons take a ticket with a number to await their turn to be served. Fetch-add-one allows an unlimited number of tickets to be issued simultaneously.

(7) Complementing fetch-add-one, is the decrementing parameter -1. Let D be the text "`fetchadd(s,-1,r1) & ... & fetchadd(s,-1,rn)`"

$$\mathfrak{M}_{\underline{i}}[\![D]\!] \equiv \begin{array}{l} \mathfrak{M}_{\text{end}(i)}[\![s]\!] = \mathfrak{M}_{\text{start}(\underline{i})}[\![s]\!] - n \\ \exists P \leftrightarrows (1,\ldots,n) \mid \forall j \in 1..n \mid \mathfrak{M}_{\text{end}(i)}[\![r_j]\!] = \mathfrak{M}_{\text{start}(\underline{i})}[\![s]\!] - P^{-1}(j) + 1 \end{array}$$

Z.6.13.2 Fetch-And Fetch-Or Fetch-Xor

Logical operations can be combined too. However, until a need is found, they will be unimplemented.

Z.6.13.3 Swap

Dynamic data structures can be concurrently manipulated with swap acting on pointers, or references. However, reference types were deliberately omitted from the type system for BLESS ! Therefore, swap is in the grammar, but its use is uncertain, and currently unimplemented.

Chapter Z.7

Behavior Expression

Z.7.1 Value

(1) For threads, `value` has all the options as subprograms[1] plus port values, a test of the current mode, and reference to property constants for this component.

(2) Values are evaluated from incoming ports and parameters, local variables, referenced data subcomponents, as well as port count, port fresh, and port dequeue.[2]

```
value ::=
  now | tops | timeout null |
  | in mode ( { mode_identifier }+ )
  | value_constant
  | variable_name
  | function_call
  | port_value
```

now the present instant with type time

tops time-of-previous-suspension

timeout AADL runtime service for hybrid dispatch protocol threads: (now-tops) ≤ Timing_Properties::Period

in mode is true when AADL mode is among those listed; false otherwise

value_constant defined in §Z.7.2, Value Constant

***variable*_name** defined in §Z.7.3, Name

function_call defined in §Z.7.7, Function Invocation

[1] see §Z.10.3
[2] BA D.7(3)

port_value defined in §Z.7.8, Port Value

Z.7.2 Value Constant

(1) Value constants are Boolean, numeric or string literals, property constants or property values.[3]

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

(2) Numeric literals are defined in §2.4 Numeric Literals. String literals are defined in §2.5 String Literals.

Semantics

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

Z.7.2.1 Property Constant

(1) Property constants are values that are defined in AADL property sets.[4]

```
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.

Z.7.2.2 Property Reference

(1) Property values may be defined in property sets, or attached to a component or feature.[5]

```
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.

[3] BA D.7(4)
[4] AS5506B §11.1.3 Property Constants
[5] BLESS Differs from BA: no local variable properties

```
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.[6]

(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.[7]

(6) When a property is an array, elements of the property value can be referenced using an integer value between brackets.[8]

```
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 ]
```

Z.7.3 Name

(1) A *name* is a sequence of identifiers, with optional array indices, separated by periods. Section §Z.8, Types, defines the relationship between names and elements of values having constructed types: arrays, records, and variants. A slice, or portion of an array, may be named by an integer-valued range as its array index.

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

(2) An array index must be an integer-valued expression (§Z.7.4), 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.

[6] BA D.7(9)
[7] BA D.7(10)
[8] BA D.7(11)

(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,[9] 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)*

Z.7.4 Expression

(1) An *expression* defines a value derived from other values by numeric or boolean operations. The type of subexpressions must be compatible with the expression's operator. The conditional boolean operators, `and then` and `or else`, demand evaluation of its left-side subexpression before its right-side subexpression, which is then evaluated only if it makes a difference to the result.[10] The other numeric and boolean operators have customary meanings.[11]

(2) Expressions have been defined to perform calculations with the complexity of programming languages such as Ada.[12] This expression language is derived from ISO/IEC 8652:1995(E), Ada95 Reference Manual §4.4.[13]

[9] 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.

[10] BA R.7(12)

[11] BA D.7(1)

[12] BA D.7(2)

[13] BA D.7(5)

```
expression ::= subexpression
    [ { + numeric_subexpression }+
    | { * numeric_subexpression }+
    |   - numeric_subexpression
    |   / numeric_subexpression
    |   mod natural_subexpression
    |   rem integer_subexpression
    |   ** numeric_subexpression
    | { and boolean_subexpression }+
    | { or boolean_subexpression }+
    | { xor boolean_subexpression }+
    |   and then boolean_subexpression
    |   or else boolean_subexpression  ]
```

Legality Rules

(L1) Operators have no precedence; parentheses must disambiguate operator order.[14]

(L2) Operands of logical operators must be boolean.[15]

(L3) Operands of numeric operators must be numeric.[16]

Semantics

(S1) Where `e` and `f` are numeric-valued expressions, and A and B are boolean-valued expressions:

$\mathfrak{M}_i[\![\texttt{e+f}]\!] \equiv \mathfrak{M}_i[\![e]\!] + \mathfrak{M}_i[\![f]\!]$ *(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)*
$\mathfrak{M}_i[\![\texttt{e}\ \textbf{mod}\ \texttt{f}]\!] \equiv \mathfrak{M}_i[\![e]\!] \bmod \mathfrak{M}_i[\![f]\!]$ *(the meaning of mod is modulus)*
$\mathfrak{M}_i[\![\texttt{e}\ \textbf{rem}\ \texttt{f}]\!] \equiv \mathfrak{M}_i[\![e]\!] - (\mathfrak{M}_i[\![f]\!] \times (\mathfrak{M}_i[\![e]\!] \div \mathfrak{M}_i[\![f]\!]))$ *(the meaning of rem is remainder)*[17]
$\mathfrak{M}_i[\![A\ \textbf{and}\ B]\!] \equiv \mathfrak{M}_i[\![A]\!] \wedge \mathfrak{M}_i[\![B]\!]$ *(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)*
$\mathfrak{M}_i[\![A\ \textbf{and then}\ B]\!] \equiv \begin{array}{l} \mathfrak{M}_i[\![A]\!] \rightarrow \mathfrak{M}_i[\![B]\!] \\ \neg\mathfrak{M}_i[\![A]\!] \rightarrow \bot \end{array}$ *(second term not evaluated if first term is false)*[18]
$\mathfrak{M}_i[\![A\ \textbf{or else}\ B]\!] \equiv \begin{array}{l} \mathfrak{M}_i[\![A]\!] \rightarrow \top \\ \neg\mathfrak{M}_i[\![A]\!] \rightarrow \mathfrak{M}_i[\![B]\!] \end{array}$ *(second term not evaluated if first term is true)*[19]

[14] BLESS Differs from BA: operator precedence
[15] BA D.7(L3)
[16] BA D.7(L5)
[17] **Reconciliation:** rem
[18] **Reconciliation:** and then
[19] **Reconciliation:** or else

Z.7.5 Subexpression

(1) A *subexpression* allows negation, complement and grouping with parentheses, or is a conditional expression (§Z.7.6).

```
subexpression ::= [ - | not | abs ]
   ( value | ( expression_or_relation ) | conditional_expression )

expression_or_relation ::=
   subexpression [ relation_symbol subexpression ]
```

Semantics

(S1) Where `e` is a numeric-valued expression, `A` is a boolean-valued expression, and c, d are expressions:

$\mathfrak{M}_i[\![\texttt{-e}]\!] \equiv 0 - \mathfrak{M}_i[\![\texttt{e}]\!]$ *(the meaning of - is negation)*
$\mathfrak{M}_i[\![\texttt{abs e}]\!] \equiv \mathfrak{M}_i[\![\texttt{(if e>=0 then e else -e)}]\!]$ *(the meaning of **abs** is absolute value)*[20]
$\mathfrak{M}_i[\![\texttt{not A}]\!] \equiv \neg\mathfrak{M}_i[\![\texttt{A}]\!]$ *(the meaning of **not** is complement)*
$\mathfrak{M}_i[\![\texttt{(A)}]\!] \equiv \mathfrak{M}_i[\![\texttt{A}]\!]$ *(the meaning of parenthesis is its contents)*
$\mathfrak{M}_i[\![c\texttt{=}d]\!] \equiv \mathfrak{M}_i[\![c]\!] = \mathfrak{M}_i[\![d]\!]$ *(the meaning of = is equality)*
$\mathfrak{M}_i[\![c\texttt{<>}d]\!] \equiv \mathfrak{M}_i[\![c\texttt{!=}d]\!] \equiv \mathfrak{M}_i[\![c]\!] \neq \mathfrak{M}_i[\![d]\!]$ *(the meaning of != is inequality)*[21]
$\mathfrak{M}_i[\![c\texttt{<}d]\!] \equiv \mathfrak{M}_i[\![c]\!] < \mathfrak{M}_i[\![d]\!]$ *(the meaning of < is less than)*
$\mathfrak{M}_i[\![c\texttt{>}d]\!] \equiv \mathfrak{M}_i[\![c]\!] > \mathfrak{M}_i[\![d]\!]$ *(the meaning of > is greater than)*
$\mathfrak{M}_i[\![c\texttt{<=}d]\!] \equiv \mathfrak{M}_i[\![c]\!] \leq \mathfrak{M}_i[\![d]\!]$ *(the meaning of <= is at most)*
$\mathfrak{M}_i[\![c\texttt{>=}d]\!] \equiv \mathfrak{M}_i[\![c]\!] \geq \mathfrak{M}_i[\![d]\!]$ *(the meaning of >= is at least)*

Z.7.6 Conditional Expression

(1) A *conditional expression* determines the value of an expression by evaluating a boolean expression or relation, then choosing between alternative expressions, returning the first if true or the second if false. At the suggestion of Jerome Hugues, Ada-style conditional expressions were added for BA2015.[22]

```
conditional_expression ::=
   ( boolean_expression_or_relation ?? expression : expression )
```

Semantics

(S1) Where `t` and `f` are expressions and `B` is a boolean-valued expression or relation:

$$\mathfrak{M}_i[\![\texttt{(if B then t else f)}]\!] \equiv \mathfrak{M}_i[\![\texttt{(B??t:f)}]\!] \equiv \begin{array}{l} \mathfrak{M}_i[\![\texttt{B}]\!] \rightarrow \mathfrak{M}_i[\![\texttt{t}]\!] \\ \neg\mathfrak{M}_i[\![\texttt{B}]\!] \rightarrow \mathfrak{M}_i[\![\texttt{f}]\!] \end{array}$$

(choose first expression if true; second expression if false)

[20] **Reconciliation:** absolute value
[21] **Reconciliation:** inequality
[22] **Reconciliation:** conditional expression

Examples

```
(if SensorConnected? and not MotionArtifact? then SpO2? else 0)
(lrl<(dn_siri<down??dn_siri:down)??lrl:(dn_siri<down??dn_siri:down))
```

Z.7.7 Function Invocation

(1) A *function call* is the invocation of a subprogram having a special form.[23] AADL subprogram components that may be invoked as functions must have one **out** parameter, preceded by any number of **in** parameters.

```
subprogram f
features
  p1 : in parameter t1;
    .
    .
    .
  pk : in parameter tk;
  result : out parameter resultType;
annex Action {** . . . **};
end mul;
```

(2) The identifiers of the formal parameters in a function call, correspond to the **in** parameters of the AADL subprogram component. Subprograms invoked as functions must use formal-actual pairs as arguments. These substitutions of actual for formal parameters are applied to the subprogram's pre- and post-conditions when verification conditions for function invocation are generated.

(3) Subprograms in other packages may be invoked as functions by prefacing their identifier with package identifiers separated by double colons.[24]

```
function_call ::= { package_identifier :: }*
   function_identifier ( [ function_parameters ] )

function_parameters ::=
  formal_expression_pair { , formal_expression_pair }*

formal_expression_pair ::= formal_identifier => actual_expression
```

Semantics

(S1) Where C is the name of a function having formal parameters $f_1, \ldots, f_k$, and $e_1, \ldots, e_k$ are expressions:

[23] Ordinarily, function calls cannot be used within expressions, because AADL doesnt have a pure function, distinct from its subprogram classifier. Because an AADL subprogram is not limited to determining its return value solely from passed values, evaluation of AADL subprograms may have side effects. Functions are AADL subprograms that are purely functional. Then function calls can be used in expressions within BA2015.

[24] **Reconciliation:** removed $ from function invocation

$\mathfrak{M}_i[\![C(f_1 \Rightarrow e_1, \ldots, f_k \Rightarrow e_k)]\!] \equiv \mathfrak{M}[\![C]\!](f_1 \Rightarrow \mathfrak{M}_i[\![e_1]\!], \ldots, f_k \Rightarrow \mathfrak{M}_i[\![e_k]\!])$
(the meaning of a function call, is the meaning of its name, applied to the meanings of its parameters)

Legality Rule

(L1) Subprograms invoked as functions must have all but the last parameter an `in` parameters, and the last parameter must be an `out` parameter.

(L2) Subprograms invoked as functions must be side-effect free; their only result is the value returned.

Examples

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

```
data number
end number;

subprogram mul
features
  --this is in the special form for a subprogram to be a function,
  --single out parameter preceded by any number of in parameters
  --may invoked within expressions as mul(e1,e2),
  --or actions as mul(v1,v2,v3)
  x : in parameter number;
  y : in parameter number;
  z : out parameter number;
annex Action {**
  post z=x*y  --postcondition relating result to values of inputs
  { z := x*y <<z=x*y>> }
**};
end mul;

subprogram cube
features  --cube is also a function
  x : in parameter number;
  y : out parameter number;
      --features other than parameters follow the out parameter
  mul : requires subprogram access mul;
annex Action {**
  post  y=x*x*x
  variables --existential quantification introduces local variables
    tmp : number;
  { mul(x,x,tmp) --invoke mul as subprogram
    ; <<tmp=x*x>>   --sequential composition
    y := mul(tmp,x) <<y=x*x*x>> } --invoke mul as function
**};
end cube;
```

Z.7.8 Port Value

(1) The core language defines that data from data ports is made available to the application source code (and Behavior_Specification) through a port variable with 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] and in the Behavior_Specifcation via functions.[25]

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

(2) The meaning of port values is defined in Z.5.5, In Event Data Ports.

[25] BLESS Differs from BA: port names must have suffix: ? or '

Chapter Z.8

Type

Z.8.1 Ideal Types

(1) The AADL core language forces subprogram parameters to be some kind of data. The core grammar allows either data types, or data implementations.

(2) The AADL Data Modeling Annex[1] defines data component classifiers that express the type and representation of values exchanged by active AADL components. This allows interoperability between languages, operating systems, hardware architectures etc. Definitions of BLESS types include Data Modeling Annex equivalents.

(3) SAE International Document AS5506B defines property types in section §11.1.1. BLESS types are equivalent to AADL property types, removing "aadl" from its reserved word to get the BLESS equivalent. Often, values defined as AADL properties need to be used in behaviors and specifications. The equivalence between BLESS and AADL property types make type checking of AADL properties used in BLESS programs straightforward.

Table Z.8.1: AADL and BLESS Type Equivalences

AADL Type	BLESS Type
`aadlreal`	`real`
`aadlinteger`	`integer`
`aadlboolean`	`boolean`
`aadlstring`	`string`

BLESS also has ideal types for `natural`, non-negative integers, `rational`, a ratio of integers, `time`, real number restricted to a type of time, and `complex`, a pair of reals.

[1] SAE International Document AS5506/2, January 2011

Z.8.2 Types are Sets

(1) A *type* is a set of values. The universe of all values, V, contains all simple values like integers and strings, and all compound values like arrays, records, and variants. A type is a set of elements of V. Moreover when ordered by set inclusion, V forms a lattice of types. The top of this lattice is the set of all values or V itself. The bottom of the lattice is the empty set. The types used by any programming language is only a small subset of V. This chapter defines a type expression language, and mapping from type expressions to sets of values.

(2) Since types are sets, subtypes are subsets. Moreover the semantic assertion"T_1 is a subtype of T_2" corresponds to the mathematical condition $T_1 \subseteq T_2$ in V. Subtyping in the basis for type checking.

Z.8.3 BLESS Type Grammar

(1) BLESS uses simple grammar to express simple, constructed types. All persistent values for variables will be statically mapped to memory addresses. No heap is needed. Stack frames will have fixed known size. Recursion prohibition limits stack depth. Much of the safety of BLESS -controlled systems comes from locking-down the type system.

Type expressions may be:

name reference to an AADL data component type having `BLESS ::Typed` property.

number **`natural`**, **`integer`**, **`rational`**, **`real`**, **`complex`**, with optionally a range, a unit, or both.

enumeration set of identifier labels

array set of elements indexed by natural number(s)

record set of labelled elements

variant one element from a set of elements determined by an identifier discriminator

boolean either **`true`** or **`false`**

string sequence of characters, §2.5

```
type ::= data_component_name | number_type | enumeration_type
  | array_type | record_type | variant_type | boolean | string
```

(2) BLESS has no unit type. Therefore unit types in BLESS must be declared as AADL property types.

(3) An AADL package is provided, **`BLESS_Types`** that extend those in `Base_Types` package defined in the Data Model Annex document. In particular, **`BLESS_Types`** have a **`BLESS_Properties::`**`Supported_Operators` list of operator symbols for types that support arithmetic. Similarly, a **`BLESS_Properties::`**`Supported_Relations` list of relation symbols defines what relations can be applied to the type. More information about **`BLESS_Types`** and **`BLESS_Properties`** can be found in Chapter §3 BLESS Package and Properties.

Z.8.4 Data Components as Types

(1) A type may refer to a data component. Data components in other packages may be referenced by a *sequence of*[2] package identifiers separated by double colon. Implementation names are formed by suffixing an identifier to the name of the data component implemented separated by a period.[3]

```
data_component_name ::= { package_identifier :: }* data_component_identifier
    [ . implementation_identifier ]
```

Legality Rule

(L1) A type name must refer to a visible data component.

Semantics

(S1) The meaning of a type name is the `BLESS ::Typed` property of the data component to which it refers.

Example

```
data ResponseFactor  --to motion
  properties
    \lang::Typed=>"integer 1..16";
end ResponseFactor;
```

Z.8.5 Enumeration Type

(1) An *enumeration* type is a sequence of identifiers. Enumeration types are expressed as the reserved word **enumeration** followed by a sequence of identifiers enclosed in parentheses.

```
enumeration_type ::= enumeration
    ( defining_enumeration_literal_identifier
    { , defining_enumeration_literal_identifier }* )
```

Property Type

(2) The AADL property type equivalent to **enumeration** `(a b c)` is **enumeration** `(a b c)`.

Data Model

(3) The Data Model equivalent to **enumeration** `(a b c)` is

```
data EnumType
  Data_Model::Data_Representation => Enum;
  Data_Model::Enumerators => ("a", "b", "c");
end EnumType;
```

[2] **Reconciliation:** multiple identifier package names

[3] In AADL grammar, an italicized prefix of a component name is merely descriptive.

and then using `EnumType` in its place, prefaced by its package name if declared in a different package.

(4) In general, where s is a sequence of identifiers separated by spaces, s' is that same sequence of identifiers enclosed in double quotes separated by commas, N is an data component identifier, and P is a package prefix so that `P::N` is a legal type name, **`enumeration`** $(s) \equiv P :: N$ such that in package P there is,

```
data N
  Data_Model::Data_Representation => Enum;
  Data_Model::Enumerators => (s');
end N;
```

Example

```
data Alarm_Type
  properties
    \lang::Typed=>"enumeration (Pump_Overheated,Defective_Battery,Low_Battery,
      POST_Failure,RAM_Failure,ROM_failure,CPU_Failure,Thread_Monitor_Failure,
      Air_In_Line,Upstream_Occlusion,Downstream_Occlusion,Empty_Reservoir,
      Basal_Overinfusion,Bolus_Overinfusion,Square_Bolus_Overinfusion,No_Alarm)";
    Data_Model::Data_Representation => Enum;
    Data_Model::Enumerators => ("Pump_Overheated","Defective_Battery","Low_Battery",
        "POST_Failure","RAM_Failure","ROM_failure","CPU_Failure","Thread_Monitor_Failure",
        "Air_In_Line","Upstream_Occlusion","Downstream_Occlusion","Empty_Reservoir",
      "Basal_Overinfusion","Bolus_Overinfusion","Square_Bolus_Overinfusion","No_Alarm");
end Alarm_Type;
```

Z.8.6 Number Type

(1) A *number type* is the name of a data component that behaves like an indivisible number, possibly restricted to a subrange, and may have units. The **time** type is equivalent to **real**, but restricted to time units.

```
number_type ::= ( natural | integer | rational | real | complex | time )
   [ constant_number_range ] [ units aadl_unit_literal_identifier ]

constant_number_range ::=
  [ [-] numeric_constant .. [-] numeric_constant  ]

numeric_constant ::= numeric_literal | numeric_property
```

(2) Number types may be restricted to a range.

Legality Rules

(L1) A number type name (its component classifier reference) must have a corresponding data component.

(L2) The upper and lower bounds of a range must have the same type as that named.

(L3) A **time** type may only have units defined by `AADL_Project::Time_Units`: ps, ns, us, sec, min, hr.

Naming Rule

(N1) A unit identifier must correspond to an AADL property unit type.

Semantics

(S1) Number types are sets (§1.1):

natural $\equiv \mathbb{N}_0$ denotes the set of all natural numbers, including 0;

integer $\equiv \mathbb{Z}$ denotes the set of all integers;

rational $\equiv \mathbb{Q}$ denotes the set of rational numbers;

real $\equiv \mathbb{R}$ denotes the set of real numbers;

complex $\equiv \mathbb{C}$ denotes the set of complex numbers;

time $\equiv \mathbb{R}$ equivalent to real, having time units.

AADL Property

(3) AADL property types for integers and real numbers have the same grammar as BLESS , except that `aadlinteger` replaces `integer`, `aadlreal` replaces `real`, and constant number ranges may have superfluous unary plus.

(4) AADL property types define a `range_type` which does not define the end points of the range. There is no equivalent to this in BLESS ; number types restricted to range must define range bounds.

Data Model

BLESS types are pure types, with unbounded magnitude. These are the closest (finite) Data Model representations.

natural `Base_Types::Natural`

integer `Base_Types::Integer`

rational `Base_Types::Float`

real `Base_Types::Float`

time `Timing_Properties::Time` (predeclared AADL property type)

complex

```
data Complex
  properties
  Data_Model::Data_Representation => Struct;
  Data_Model::Base_Type => (classifier(Base_Types::Float), classifier(Base_Types::Float));
  Data_Model::Element_Names => ("re", "im");  --real and imaginary parts
end Complex;
```

Z.8.7 Array Type

(1) An *array type* is a collection indexed by natural numbers. The natural numbers in an array type expression denote the size of the array in successive dimensions. The sizes may be expressed as natural literals, or identifiers of natural number values.[4]

```
array_type ::= array [ array_range_list ] of type

array_range_list ::= natural_range { , natural_range }*

natural_range ::= natural_number [ .. natural_number ]

natural_number ::=
 natural_integer_literal
 | natural_constant_identifier
 | natural_property
```

Legality Rule

(L1) For all ranges of natural numbers a and b, used to define ranges $a..b$, a must be at most b, $a \leq b$.

Data Model

(2) The Data Model for arrays uses the property `Data_Model::Slice` to define ranges for each array dimension rather than the property `Data_Model::Dimension` which only defines the array size. An single integer literal array dimension is interpreted as a range from zero. The Data Model equivalent to `array [5, 0..15, May..October] of MyPackage::MyElementType` is

```
data My_Three_Dimensional_Array
  properties
  Data_Model::Data_Representation => Array;
  Data_Model::Base_Type => (classifier (MyPackage::MyElementType));
  Data_Model::Slice => (0..4,    --5 becomes 0..4
    0..15,  --same range of natural literals
    May..October);  --May and October must identify natural values
end My_Three_Dimensional_Array
```

(3) In general, where n is a sequence of positive integer literals, integer ranges (i.e. 1..10), n' is that same sequence separated by commas having single integer literals replaced by integer ranges starting at zero, and E and T are data component identifiers, and P and R are package prefixes so that `P::T` and `R::E` are legal type names, `array` $[n]$ `of` $R :: E \equiv P :: T$ such that in package P there is,

```
data T
  Data_Model::Data_Representation => Array;
  Data_Model::Base_Type => (classifier (R::E));
  Data_Model::Slice => (n');
end T;
```

[4] Enumeration types for array indices were removed in v0.13 June 2010. Negative array indices are thus disallowed.

(4) The Data Model also allows `Data_Model::Dimension` to be used which may only be a list of integer literals. The equivalent array type uses the same list without commas

Example

```
data Fault_Log  --holds records of faults
  properties
    \lang::Typed => "array [PCA_Properties::Fault_Log_Size] of PCA_Types::Fault_Record";
    Data_Model::Data_Representation => Array;
    Data_Model::Base_Type => (classifier(Fault_Record));
    Data_Model::Dimension => (PCA_Properties::Fault_Log_Size);
end Fault_Log;
```

Z.8.8 Record Type

(1) A *record type* is a collection of types indexed by identifier labels.

```
record_type ::= record ( { record_field }+ )
record_field ::= defining_field_identifier : type ;
```

Data Model

The Data Model equivalent to `record ( l1:T1; l2:T2; )` is

```
data My_Record
  properties
  Data_Model::Data_Representation => Struct;
  Data_Model::Base_Type => (classifier (T1), classifier (T2));
  Data_Model::Element_Names => ("l1", "l2");
end My_Record;
```

(2) In general, where S is a sequence of pairs of labels and type names, where each label is separated from its type name by a colon and followed by a semicolon,[5] B is a sequence of the second elements of those pairs (type names) of S enclosed in parentheses prefaced by `classifier` separated by commas,[6] and L is a sequence of the first elements of those pairs (labels) of S enclosed in double-quotes and separated by commas,[7] and P is package prefix so that `P::T` is a legal type name, `record` $(S) \equiv P::T$ such that in package P there is,

```
data T
  Data_Model::Data_Representation => Struct;
  Data_Model::Base_Type => (B);
  Data_Model::Element_Names => (L);
end T;
```

[5] i.e. l1:T1; l2:T2; l3:T3
[6] i.e. classifier (T1), classifier (T2), classifier(T3)
[7] i.e. “l1”, “l2”, “l3”

(3) The Data Model Annex shows an alternate way to represent records (structs) using subcomponents of data component implementations to represent record elements. These are not supported by BLESS . Use the Data Model properties instead.

Example

```
data Fault_Record  --record of fault for log
  properties
   BLESS::Typed => "record (alarm:Alarm_Type; warning:Warning_Type;
        occurrence_time:BLESS_Types::Time;)";
    Data_Model::Data_Representation => Struct;
    Data_Model::Element_Names => ("alarm","warning","occurrence_time");
    Data_Model::Base_Type => ( classifier(Alarm_Type),classifier(Warning_Type),
        classifier(BLESS_Types::Time));
end Fault_Record;
```

Z.8.9 Variant Type

A *variant type* holds a value of varying type specified by the value of a discriminant. A discriminant holds the value of one of the labels of the record fields, which then determines the type of the variant.

```
variant_type ::= variant [ discriminant_identifier ]
    ( { record_field }+ )
```

Legality Rules

(L1) A value of variant type may only have its discriminant set at creation; discriminants may never be the subject of assignment.

(L2) A value of variant type has the type indicated by its discriminant; accessing that value as any other type is an error.

Data Model

The Data Model equivalent to `variant [d] (c1:T1; c2:T2;)` is

```
data My_Variant
  properties
  Data_Model::Data_Representation => Union;
  Data_Model::Base_Type => (classifier (T1), classifier (T2) );
  Data_Model::Element_Names => ("c1", "c2" );
end My_Variant
```

(1) In general, where S is a sequence of pairs of labels and type names, where each label is separated from its type name by a colon and followed by a semicolon,B is a sequence of the second elements of those pairs (type names) of S enclosed in parentheses prefaced by `classifier` separated by commas,and L is a sequence of the first elements of those pairs (labels) of S enclosed in double-quotes and separated by commas,d is a discriminant identifier, and P is package prefix so that `P::T` is a legal type name, `variant [d] (`S`)` $\equiv$ $P :: T$ such that in package P there is,

```
data T
  Data_Model::Data_Representation => Union;
  Data_Model::Base_Type => (B);
  Data_Model::Element_Names => (L);
end T;
```

(2) The Data Model Annex shows an alternate way to represent variants (unions) using subcomponents of data component implementations to represent record elements. These are not supported by BLESS . Use the Data Model properties instead.

Example

```
data Event_Record  --record of event for log
  properties
    BLESS::Typed => "variant (start_patient_bolus:Start_Patient_Bolus_Event;
      stop_patient_bolus:Stop_Patient_Bolus_Event;)";
    Data_Model::Data_Representation => Union;
    Data_Model::Base_Type => (classifier (Start_Patient_Bolus_Event),
      classifier (Stop_Patient_Bolus_Event));
    Data_Model::Element_Names => ("start_patient_bolus", "stop_patient_bolus" );
end Event_Record;
```

Z.8.10 Type Inclusion Rules

A type is included in another type $t \subseteq s$ when every value of one type is also a value of the other. $t \subseteq s \equiv \forall v \in t | v \in s$

In the following type rules,

- type expressions are denoted by s, t, and u,
- $s \rightarrow t$ is a function with domain s and range t;[8]
- type names by a and b, and element labels by L;
- V is the set of all values;
- d is a discriminant label;
- C is a set of inclusion constraints for types;
- $C.a \subseteq b$ is the set C extended with the constraint that type a is included in b;
- $C \models t \subseteq s$ is an assertion that from C we can infer $t \subseteq s$.

[TOP]: $C \models t \subseteq \mathrm{V}$ *(every type is included in the set of all values)*

[8] see §Z.7.7 Function Invocation for the form of AADL subprograms to be used as a function by BLESS . For functions with k parameters, s is a tuple of types $(s_1, \ldots, s_k)$.

[VAR]: $C.a \subseteq t \models a \subseteq t$ *(what it means to extend a type constraint)*

[BAS]: $C \models a \subseteq a$ *(every type includes itself)*

[TRANS]: $$\frac{C \models s \subseteq t \ \wedge \ C \models t \subseteq u}{C \models s \subseteq u}$$ *(type inclusion is transitive)*

[FUN]: $$\frac{C \models s \subseteq s_1 \wedge C \models t \subseteq t_1}{C \models (s \rightarrow t) \subseteq (s_1 \rightarrow t_1)}$$ *(a function type includes another when its domain includes the other's domain and its range includes the other's range)*

[CAR]: $$\frac{C \models s \subseteq t \wedge n \leq m}{C \models \texttt{array}[n] \texttt{ of } s \subseteq \texttt{array}[m] \texttt{ of } t}$$

(an array type includes another when its element type includes the other's element type, and the other has at most as many elements)

[CARM]: $$\frac{C \models s \subseteq t}{C \models \texttt{array}[n_1, n_2, \ldots, n_k] \texttt{ of } s \subseteq \texttt{array}[n_1, n_2, \ldots, n_k] \texttt{ of } t}$$

(a multi-dimensional array includes another when its element type includes the other's element type, and has exactly the same dimensions)

[SLICE]: $$\frac{C \models s \subseteq t \wedge d \leq a \wedge b \leq e}{C \models \texttt{array}[a..b] \texttt{ of } s \subseteq \texttt{array}[d..e] \texttt{ of } t}$$

(an array slice includes another when its element type includes the other's element type, and its range includes the other's range)

[SLICEM]: $$\frac{C \models s \subseteq t \wedge \forall i \in \{1, ..., k\} | d_i \leq a_i \wedge b_i \leq e_i}{C \models \texttt{array}[a_1..b_1, \ldots, a_k..b_k] \texttt{ of } s \subseteq \texttt{array}[d_1..e_1, \ldots, d_k..e_k] \texttt{ of } t}$$

(a multi-dimensional slice includes another when its element type includes the other's element type, and for each dimension its range includes the other's range)

[RECD]: $$\frac{C \models s_1 \subseteq t_1 \wedge \cdots \wedge C \models s_n \subseteq t_n}{C \models \texttt{record}\,(L_1 : s_1;\ \ldots\ L_n : s_n; \ldots L_m : s_m;) \subseteq \texttt{record}\,(L_1 : t_1;\ \ldots\ L_n : t_n;)}$$

(a record type includes another when the other has elements the same labels, and perhaps additional others, and for each label the corresponding element type includes the other's element type for that label)

$$\text{[VART]: } \frac{C \models s_1 \subseteq t_1 \wedge \cdots \wedge C \models s_n \subseteq t_n}{C \models \textbf{variant}(L_1 : s_1;\ \ldots\ L_n : s_n;) \subseteq \textbf{variant}(L_1 : t_1;\ \ldots\ L_n : t_n;)}$$

(a variant type includes another when the other has elements the same labels, and for each label the corresponding element type includes the other's element type for that label)

Z.8.11 Type Rules for Expressions

(1) Type rules for expressions determine types of expressions, especially complex names.

(2) Relation symbols, `= !=`, are treated as functions of pairs of the same element type to **boolean**, $(s, s) \rightarrow$ **boolean**, and are defined for every type s.

Relation symbols, `< <= >= >`, are treated as functions of pairs of the same element type to **boolean**, $(s, s) \rightarrow$ **boolean**, and are pre-defined for types **natural integer rational real complex**.

(3) Numeric operator symbols, `+ *`, are treated as functions of sequences of the same element type to that element type, $(s, \ldots, s) \rightarrow s$, and are pre-defined for types **natural integer rational real complex**.

Numeric operator symbols, `- / mod rem **`, are treated as functions of pairs of the same element type to that element type, $(s, s) \rightarrow s$, and are pre-defined for types **natural integer rational real complex**.

Unary – is arithmetic negation, $s \rightarrow s$, and is pre-defined for types **integer rational real complex**.

(4) Logical operator symbols, `and or xor`, are treated as functions of sequences of **boolean** to **boolean**, (**boolean**, . . . ,**boolean**) → **boolean**.

Logical operator symbols, `cand cor`, are treated as functions of pairs of **boolean** to **boolean**, (**boolean**,**boolean**) → **boolean**.

Unary **not** is complement, **boolean** → **boolean**.

(5) In the following type rules,

A is a set of type assumptions for variables;

C is a set of inclusion constraints for types;

$\mathbf{V}$ is the set of all values;

e is an expression;

s, t are types;

$s \rightarrow t$ is a function with domain s and range t;[9]

x is a variable;

L is a field label;

d is a discriminant label;

$A.x : t$ is the set A extended with the assumption that variable x has type t;

$C, A \models e : t$ means that from the set of constraints C and the set of type assumptions A, we can infer that expression e has type t;

$f : s \rightarrow t$ means f is a function with domain type s and range type t:[10]

```
subprogram f features x:in parameter s; y:out parameter t; end f;
```

[ETOP]: $C, A \models e : \mathrm{V}$ *(the type of every expression is included in the set of all values)*

[EVAR]: $C, A.e : t \models e : t$ *(define extending a type assumption)*

[ETRANS]: $$\frac{C, A \models e : t \ \wedge \ C \models t \subseteq u}{C, A \models e : u}$$ *(type inclusion is transitive for expressions too)*

[APPL]: $$\frac{C, A \models f : s \rightarrow t \wedge C, A \models x : s}{C, A \models f(x) : t}$$ *(a function of type $s \rightarrow t$, applied to a parameter with type s, has type t)*

[ECAR]: $$\frac{C \models x : \texttt{array}[n] \ \texttt{of} \ s \ \wedge \ 0 \leq m < n}{C \models \ x[m] : s}$$ *(indexing a variable of array type has the array's element type)*

[ECARM]: $$\frac{C \models x : \texttt{array}[n_1, n_2, \ldots, n_k] \ \texttt{of} \ s \quad 0 \leq m_1 < n_1 \wedge \cdots \wedge 0 \leq m_k < n_k}{C \models \ x[m_1, m_2, \ldots, m_k] : s}$$ *(indexing a variable of multi-dimensional array type has the array's element type)*

[SEL]: $$\frac{C, A \models x : \texttt{record}(L_1 : t_1; \ldots L_n : t_n;)}{C, A \models x.L_i : t_i \quad i \in 1..n}$$ *(selecting a label of a variable having record type, has the type of the labeled element)*

[9] For functions with k parameters, s is a tuple of types $(s_1, \ldots, s_k)$.

[10] see §Z.7.7 Function Invocation for the form of AADL subprograms to be used as a function by BLESS .

[VSEL]: $\dfrac{C, A \models x : \texttt{variant}[d](L_1 : t_1; \ldots L_n : t_n;)}{C, A \models x.L_i : t_i \text{ iff } x.d = L_i \quad i \in 1..n}$ *(selecting a label of a variable having variant type, has the type of the labeled element, only when the label is same as the discriminant)*

Chapter Z.9

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.

Z.9.1 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)>>
**};
```

Z.9.2 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 ) >>
```

Z.9.2.1 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
```

Z.9.2.2 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 Z.9.3.5, 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.

Z.9.2.3 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 Z.9.4.6, 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)>>
```

Z.9.2.4 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 Z.9.4.7, 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 >>
```

Z.9.3 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 Z.9.3.8, and existential quantification is defined in D Z.9.3.9.

```
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:

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

Example

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

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

Z.9.3.1 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 Z.9.3.5.

(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:

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

Z.9.3.2 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 (Z.9.3.3) and period-shift (D Z.9.3.4) are time-free (e.g. no `'` `@` or `^` within). Grammar and meaning of a name is defined in Z.7.3 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))>>
```

Z.9.3.3 Time-Expression

(1) Both timed predicate (Z.9.3.2 Timed Predicate) and timed expression (Z.9.4.1 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),

$\mathfrak{M}_t[\![\texttt{e+f}]\!] \equiv \mathfrak{M}_t[\![e]\!] + \mathfrak{M}_t[\![f]\!]$ *(the meaning of + is addition)*
$\mathfrak{M}_t[\![\texttt{e*f}]\!] \equiv \mathfrak{M}_t[\![e]\!] \times \mathfrak{M}_t[\![f]\!]$ *(the meaning of * is multiplication)*
$\mathfrak{M}_t[\![\texttt{e-f}]\!] \equiv \mathfrak{M}_t[\![e]\!] - \mathfrak{M}_t[\![f]\!]$ *(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)*
$\mathfrak{M}_i[\![\texttt{(e)}]\!] \equiv \mathfrak{M}_i[\![e]\!]$ *(the meaning of parentheses is its contents)*
$\mathfrak{M}_i[\![\texttt{-e}]\!] \equiv 0.0 - \mathfrak{M}_i[\![e]\!]$ *(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) >>
```

Z.9.3.4 Period-Shift

(1) Both timed predicate (Z.9.3.2) and timed expression (Z.9.4.1) 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,

$\mathfrak{M}_i[\![\texttt{(e+f)}]\!] \equiv \mathfrak{M}_i[\![e]\!] + \mathfrak{M}_i[\![f]\!]$ *(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)*
$\mathfrak{M}_i[\![\texttt{(e/f)}]\!] \equiv \mathfrak{M}_i[\![e]\!] \ / \ \mathfrak{M}_i[\![f]\!]$ *(the meaning of / is division, neglecting remainder)*
$\mathfrak{M}_i[\![\texttt{-e}]\!] \equiv 0 - \mathfrak{M}_i[\![e]\!]$ *(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))))>>;
```

Z.9.3.5 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

$\mathfrak{M}_i[\![\texttt{B(f1:a1, f2:a2, ... fn:an)}]\!] \equiv \mathfrak{M}_i[\![B \mid \frac{f_1}{a_1} \mid \frac{f_2}{a_2} \cdots \mid \frac{f_n}{a_n}]\!]$
(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)>>
```

Z.9.3.6 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 (Z.7.5). 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)*
$\mathfrak{M}_i[\![c \text{ in } l..u]\!] \equiv \mathfrak{M}_i[\![c]\!] \geq \mathfrak{M}_i[\![l]\!] \wedge \mathfrak{M}_i[\![c]\!] \leq \mathfrak{M}_i[\![u]\!]$ *(the meaning of .. is closed interval)*
$\mathfrak{M}_i[\![c \text{ in } l,.u]\!] \equiv \mathfrak{M}_i[\![c]\!] > \mathfrak{M}_i[\![l]\!] \wedge \mathfrak{M}_i[\![c]\!] \leq \mathfrak{M}_i[\![u]\!]$ *(the meaning of ,. is open-left interval)*
$\mathfrak{M}_i[\![c \text{ in } l.,u]\!] \equiv \mathfrak{M}_i[\![c]\!] \geq \mathfrak{M}_i[\![l]\!] \wedge \mathfrak{M}_i[\![c]\!] < \mathfrak{M}_i[\![u]\!]$ *(the meaning of ., is open-right interval)*
$\mathfrak{M}_i[\![c \text{ in } l,,u]\!] \equiv \mathfrak{M}_i[\![c]\!] > \mathfrak{M}_i[\![l]\!] \wedge \mathfrak{M}_i[\![c]\!] > \mathfrak{M}_i[\![u]\!]$ *(the meaning of ,, is open interval)*

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

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

Z.9.3.7 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

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

Z.9.3.8 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,

$\mathfrak{M}_i[\![$`all v:T in R are P(v)`$]\!] \equiv \forall\, v \in \mathfrak{M}_i[\![R]\!] \subseteq \mathfrak{M}_i[\![T]\!] \mid \mathfrak{M}_i[\![P(v)]\!]$
(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))>>
```

Z.9.3.9 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.

$\mathfrak{M}_i[\![$**exists** v:T **in** R **that** P(v)$]\!] \equiv \exists\, v \in \mathfrak{M}_i[\![R]\!] \subseteq \mathfrak{M}_i[\![T]\!] \mid \mathfrak{M}_i[\![P(v)]\!]$
(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)))>>
```

Z.9.3.10 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 1.17 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> $\equiv \hat{p} \equiv \mathfrak{M}_{now}[\![p \neq \perp]\!]$.
Where v is a variable identifier <v> $\equiv \hat{v} \equiv \mathfrak{M}_{now}[\![v \neq \perp]\!]$.
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> $\equiv \hat{x} \hat{-} \hat{y}$.

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

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

Z.9.4 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 (Z.9.2.3) 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)
$\mathfrak{M}_i[\![$`e+f`$]\!] \equiv \mathfrak{M}_i[\![e]\!] + \mathfrak{M}_i[\![f]\!]$ *(the meaning of + is addition)*
$\mathfrak{M}_i[\![$`e*f`$]\!] \equiv \mathfrak{M}_i[\![e]\!] \times \mathfrak{M}_i[\![f]\!]$ *(the meaning of * is multiplication)*
$\mathfrak{M}_i[\![$`e-f`$]\!] \equiv \mathfrak{M}_i[\![e]\!] - \mathfrak{M}_i[\![f]\!]$ *(the meaning of - is subtraction)*
$\mathfrak{M}_i[\![$`e/f`$]\!] \equiv \mathfrak{M}_i[\![e]\!] \div \mathfrak{M}_i[\![f]\!]$ *(the meaning of / is division)*
$\mathfrak{M}_i[\![$`e**f`$]\!] \equiv \mathfrak{M}_i[\![e]\!]^{\mathfrak{M}_i[\![f]\!]}$ *(the meaning of ** is exponentiation)*
$\mathfrak{M}_i[\![$`e mod f`$]\!] \equiv \mathfrak{M}_i[\![e]\!] \bmod \mathfrak{M}_i[\![f]\!]$ *(the meaning of **mod** is modulus)*
$\mathfrak{M}_i[\![$`e rem f`$]\!] \equiv \mathfrak{M}_i[\![e]\!] \operatorname{rem} \mathfrak{M}_i[\![f]\!]$ *(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,

$\mathfrak{M}_i[\![-S]\!] \equiv 0 - \mathfrak{M}_i[\![S]\!]$ *(the meaning of - is negation)*
$\mathfrak{M}_i[\![$**abs** S$]\!] \equiv \mathfrak{M}_i[\![$(**if** S>=0 **then** S **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)))>>
```

Z.9.4.1 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))>>
```

Z.9.4.2 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
```

Z.9.4.3 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 Z.9.3.5. 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
```

Z.9.4.4 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:

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

(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)>>
```

Z.9.4.5 Conditional Assertion Function

(1) A *conditional assertion function* is much like a conditional assertion expression (Z.9.4.4), 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 (Z.9.2.3).

(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:

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

(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
>>
```

Z.9.4.6 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

$\mathfrak{M}_i[\![\texttt{C(a1 a2 ... an)}]\!] \equiv \mathfrak{M}_i[\![E \mid {}^{f_1}_{a_1} \mid {}^{f_2}_{a_2} \cdots \mid {}^{f_n}_{a_n}]\!]$
(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)>>
```

Z.9.4.7 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

$\mathfrak{M}_i[\![\texttt{C(a)}]\!] \equiv \mathfrak{M}_i[\![R(a)]\!]$

(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 Z.10

Subprogram

(1) Subprogram behavior is defined using the *Action annex sublanguage*. Only subprogram components have `Action` annexes.

```
subprogram_annex_subclause ::=
  annex Action {** subprogram_behavior **} ;
```

Z.10.1 Subprogram Behavior

(1) An `Action` annex consists of a behavior action block that may be preceded by Assertions visible in the scope of the subprogram, a precondition, and a postcondition. A precondition that must be true of the subprogram parameters is preceded by pre. A postcondition that will be true after execution of the subprogram is preceded by post.

```
subprogram_behavior ::=
   [ assert { assertion }+ ]
   [ pre assertion ]
   [ post assertion ]
   [ invariant assertion ]
   behavior_action_block
```

(2) In most programming languages, a subprogram is comprised from imperative commands that assign values of expressions to variables or control the flow of execution with branches and loops. For BAv2 subprograms, the temporal logic formula comprising the main body of the subprogram is satisfied by lattices of states (§1.9). Execution of a BAv2 subprogram constructs a satisfying state lattice. Typically many different lattices satisfy the same temporal logic formula, all of which will have identical bindings of values to variables in their start and end states.

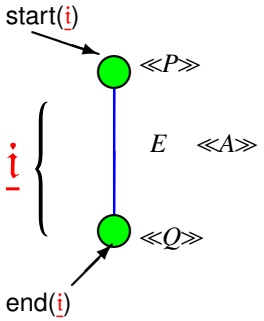

Figure Z.10.1: Subprogram Satisfying Lattice

(3) Figure Z.10.1 depicts a lattice that satisfies a subprogram having precondition P, post condition Q, by constructing an interval i, satisfying E, its existential lattice quantification. Although depicted as a single arc from the interval's start node to its end node, satisfying lattices will have many intermediate nodes and arcs.

(4) Subprograms may also assert invariants that must be true in every state. Figure Z.10.1 depicts an invariant, A that must hold in every state in i. Both E and A are logic formulas, of different logics. The difference is that e is an interval temporal logic satisfied by the combined structure of states (nodes) and transitions (arcs), while A is a first-order predicate applied to each of the states individually.

(5) Within the behavior annex subclause the value of a data component is returned by naming the data subcomponent, the requires data access, or the provides data access feature. Multiple references to this name represent multiple reads that may return different values, if the data component is shared and a write has been performed concurrently between the two reads. Concurrent writes may be prevented by a value of the `Concurrency_Control_Protocol` property that ensures mutual exclusion over an execution sequence with multiple reads.[1]

(6) A transition action can assign a return value to an outgoing parameter of the containing subprogram type by naming the parameter on the left-hand side of the assignment, i.e., `par :=v`. A transition action can assign a value to an incoming parameter of a subprogram call by specifying the value v in place of the formal parameter.[2]

Legality Rule

(L1) Assertions of subprograms must not have temporal operators `@`, `^`, or `'`.[3]

Semantics

(S1) Where A, P, and Q are predicates, and E is existential lattice quantification:

$$\mathfrak{M}_{\mathrm{i}}[\![\textbf{assert } \ll A \gg \textbf{ pre } \ll P \gg \textbf{ post } \ll Q \gg \ E]\!] \equiv \mathfrak{M}_{start(\mathrm{i})}[\![P]\!] \wedge \mathfrak{M}_{end(\mathrm{i})}[\![Q]\!] \wedge \mathfrak{M}_{\mathrm{i}}[\![E]\!] \wedge \mathfrak{M}_{\mathrm{i}}[\![A]\!]$$

(the meaning of subprogram behavior is: P is true in the stating state of i, Q is true in the ending state of i, A is true throughout i, and i satisfies E)

(S2) Equivalently, **assert <<A>> pre <<P>> post <<Q>>** `E` has the behavior of an automata transition $T(s, true, d, true)[E]$ from `initial` state s in which assertion $\ll P \gg$ holds to `final` state d in which assertion $\ll Q \gg$ holds while performing action E.

Example

```
subprogram minimum3
features
  a: in parameter BAv2_Types::Real;
  b: in parameter BAv2_Types::Real;
  c: in parameter BAv2_Types::Real;
  result: out parameter BAv2_Types::Real;
annex Action
```

[1] BA D.5(10)
[2] BA D.5(17)
[3] Without temporal operators, assertions are first-order predicates.

```
{**
assert <<MIN3:a b c:=(a<MIN(a:b,b:c) ?? a : MIN(a:b,b:c))>>
pre <<a>0 and b>0 and c>0>>
post <<result=MIN3(a:a,b:b,c:c)>>
  {
    <<true>>
  result := (c<(a < b ?? a : b) ?? c : (a < b ?? a : b))
  <<result=MIN3(a:a,b:b,c:c)>>
  }
**};
end minimum3;
```

Z.10.2 Subprogram Basic Actions

(1) Within a Action annex, the only basic actions are skip, assignment, simultaneous assignment, and exception throwing.[4] For threads `basic_action` includes other actions not performed by subprograms (§Z.6.4).

```
basic_action ::=
  skip | assignment  | simultaneous_assignment | when_throw
  | subprogram_invocation
```

Z.10.3 Value for Subprograms

(1) An *value* is indivisible and may be a variable name, a function call, a reserved word, a property constant or a literal. Literals have the same representation as the core language.[5]

```
value ::=
  variable_name | value_constant | function_call
  | incoming_subprogram_parameter_identifier | null
```

[4] **Reconciliation:** subprogram actions
[5] AS5506B §15.4 Numeric Literals

Chapter 1

Appendix: Mathematics

*

(1) To prove correctness, a programming language must be mathematically defined. Therefore, the foundational mathematics must be derived from First Principles.

The foundational mathematics was deliberately selected to be as simple as possible, using only a few fundamental concepts from which all else flows. The following sections tersely declare these fundamental concepts, and is not meant as a textbook or tutorial. Many pieces of standard mathematics, like numbers and arithmetic are just assumed.[1]

Later, computation will be defines as satisfaction of interval-temporal logic formulas with lattices of states–not as a sequence of imperative commands. A lattice is a relation with some special properties. Thinking about programs as logic formulas, instead of traditional, imperative, sequential control flow, takes some getting used to.

Still, this document attempts to be self-contained, explicitly built on simple math defined herein, starting with sets.

Appendix 1.1 Sets

(1) A *set* is a collection of elements. Finite sets may be specified by enumerating their elements between curly braces. For example, $\{true, false\}$ denotes the set consisting of the Boolean constants *true* and *false*. When enumerating elements of a set, "..." is used to denote repetition. For example, $\{1, \ldots, n\}$ denotes the set of natural numbers from 1 to n where the upper bound, n, is a natural number that is not further specified.

[1] as defined by *CRC Concise Encyclopedia of Mathematics*, Eric W Weisstein, editor, second edition, Chapman & Hall/CRC, 2003.

(2) More generally, sets are specified by referring to some property of their elements. $\{x \mid P\}$ denotes the set of all elements x that satisfy the property P. The bar, $\mid$, can be read as "such that". For example, $\{x \mid x$ is an integer and x is divisible by 2$\}$ denotes the infinite set of all even integers.

(3) For membership, $a \in A$ denotes that a is an element of the set A, and $b \notin A$ to denote that b is not an element of the set A.

(4) Some sets have customary symbols:

$\emptyset$ denotes the empty set;

$\mathbb{N}_0$ denotes the set of all natural numbers, including 0;

$\mathbb{Z}$ denotes the set of all integers;

$\mathbb{Q}$ denotes the set of rational numbers;

$\mathbb{R}$ denotes the set of real numbers;

$\mathbb{C}$ denotes the set of complex numbers.

$\mathbb{B}$ denotes the set $\{true, false\}$.

Fixed-point numbers are rational numbers with fixed divisor.

(5) In a set, one does not distinguish repetitions of elements. Thus $\{T, F\}$ and $\{T, T, F\}$ are the same set. Often it is convenient to refer to a given set when defining a new set. $\{x \in A \mid P\}$ is an abbreviation for $\{x \mid x \in A$ and $P\}$. Similarly, the order of elements is irrelevant. Two sets A and B are equal, $B = A$, if-and-only-if they have the same elements.

(6) Let A and B be sets. Then $A \subseteq B$ denotes that A is a *subset* of B; $A \cap B$ denotes the *intersection* of A and B; $A \cup B$ denotes the *union* of A and B; and, $A - B$ denotes the *difference* of A and B. The symbol $\equiv$ is used to define equivalence.

$A \subseteq B \equiv a \in B$ for every $a \in A$
$A \cap B \equiv \{a \mid a \in A$ and $a \in B\}$
$A \cup B \equiv \{a \mid a \in A$ or $a \in B\}$
$A - B \equiv \{a \mid a \in A$ and $a \notin B\}$

(7) Sets A and B are *disjoint* if they have no element in common, $A \cap B = \emptyset$.

(8) The definitions of intersection and union can be generalized to more than two sets. Let A_k be a set for every element k of some other set J in $\bigcap_{k \in J} \equiv \{a \mid a \in A_k$ for all $k \in J\}$ $\bigcup_{k \in J} \equiv \{a \mid a \in A_k$ for some $k \in J\}$

(9) For a finite set A, $\|A\|$ denotes the *cardinality*, or number of elements in A. For a non-empty, finite set $B \subset \mathbb{Z}$, $\min(B)$ denotes the *minimum* of all integers in B.

(10) $\mathbb{D}$ is the set of all possible constructed values, including strings, records, and arrays, formally defined in DZ.8 Type.

Appendix 1.2 Tuples

(1) For sets, the repetition of elements and their order is irrelevant. When ordering matters, ordered pairs and tuples are used. For elements a and b, not necessarily distinct, $\langle a, b\rangle$ is an *ordered pair* or simply pair. Then a and b are called *components* of $\langle a, b\rangle$. Two pairs $\langle a, b\rangle$ and $\langle b, c\rangle$ are equal, $\langle a, b\rangle = \langle c, d\rangle$ if-and-only-if $a = c$ and $b = d$.

(2) More generally, let n be any natural number, $n \in \mathbb{N}_0$. Then if $a_1, \ldots, a_n$ are any n elements, then $\langle a_1, \ldots, a_n\rangle$ is an *n-tuple*. The element a_k where $k \in \{1, \ldots, n\}$ is called the k-th element of $\langle a_1, \ldots, a_n\rangle$. An n-tuple $\langle a_1, \ldots, a_n\rangle$ is equal to an m-tuple $\langle b_1, \ldots, b_m\rangle$, $\langle a_1, \ldots, a_n\rangle = \langle b_1, \ldots, b_m\rangle$, if-and-only-if $m = n$ and $a_k = b_k$ for all $k \in \{1, \ldots, n\}$. Note that 2-tuples are pairs. Additionally, a 0-tuple is written as $\langle\rangle$, and a 1-tuple as $\langle a\rangle$ for any element a.

(3) The *Cartesian product*, $A \times B$ of sets A and B consists of all pairs, $\langle a, b\rangle$ with $a \in A$ and $b \in B$. The n-fold Cartesian product, $A_1 \times \ldots \times A_n$ of sets $A_1, \ldots, A_n$ consists of all n-tuples, $\langle a_1, \ldots, a_n\rangle$ with $a_k \in A_k$ for $k \in \{1, \ldots, n\}$. If all A_k are the same set A, then the n-fold Cartesian product, $A \times \ldots \times A$ is also written A^n.

Appendix 1.3 Relations

(1) A binary *relation* R between sets A and B is a subset of their Cartesian product, $A \times B$, that is, $R \subseteq A \times B$. If $A = B$, then R is called a relation on A. For example, $\{\langle a, 1\rangle, \langle b, 2\rangle, \langle c, 2\rangle\}$ is a binary relation between $\{a, b, c\}$ and $\{1, 2\}$. More generally, for any natural number n, and n-ary relation R between sets $A_1, \ldots, A_n$ is a subset of the n-fold Cartesian product $A_1 \times \ldots \times A_n$, that is, $R \subseteq A_1 \times \ldots \times A_n$. Note that 1-ary relations are called unary relations, 2-ary relations are called binary relations, and 3-ary relations are called ternary relations.

(2) Consider a binary relation R on a set A. R is called *reflexive* if $\langle a, a\rangle \in R$ for every $a \in A$; it is called *irreflexive* if $\langle a, a\rangle \notin R$ for every $a \in A$. R is called *symmetric* if for all $a, b \in R$, whenever $\langle a, b\rangle \in R$ the also $\langle b, a\rangle \in R$; it is called *antisymmetric* if for all $a, b \in A$, whenever $\langle a, b\rangle \in R$ and $\langle b, a\rangle \in R$ then $b = a$. R is called *transitive* if for all $a, b, c \in A$ whenever $\langle a, b\rangle \in R$ and $\langle b, c\rangle \in R$ then also $\langle a, c\rangle \in R$.

(3) The transitive, reflexive *closure*, $R^\star$, of a binary relation R over a set A, is the smallest, transitive and reflexive, binary relation on A that contains R as a subset. The transitive, irreflexive closure, R^+, of a binary relation R over a set A, is the smallest, transitive and irreflexive binary relation that contains R as a subset.

$$R^\star \equiv \; R \subseteq R^\star \text{ and for all } a, b, c \in A \mid \begin{array}{c} \langle a, b\rangle \in R^\star \wedge \langle b, c\rangle \in R^\star \rightarrow \langle a, c\rangle \in R^\star \\ \langle a, a\rangle \in R^\star \end{array}$$

$$R^+ \equiv \; R \subseteq R^+ \text{ and for all } a, b, c \in A \mid \begin{array}{c} \langle a, b\rangle \in R^+ \wedge \langle b, c\rangle \in R^+ \rightarrow \langle a, c\rangle \in R^+ \\ \langle a, a\rangle \notin R^+ \end{array}$$

(4) The *relational composition*, $R_1 \circ R_2$, of relations R_1 and R_2 on a set A creates a new relation by combining them:

$$R_1 \circ R_2 \equiv \{\langle a, c\rangle \mid \text{ there exists } b \in A \text{ with } \langle a, b\rangle \in R_1 \text{ and } \langle b, c\rangle \in R_2\}$$

(5) For any natural number n, the n-fold relational composition, R^n, of a relation R on a set A is defined inductively:

$R^0 \equiv \{\langle a, a\rangle \mid a \in A\}$

$R^n \equiv R^{n-1} \circ R$ for $n > 0$.

$R^\star \equiv \bigcup_{n\in\mathbb{N}_0} R^n$

$R^+ \equiv R^\star - R^0$

(6) Membership of pairs in a binary relation is usually written in infix notation; instead of $\langle a, b\rangle \in R$, use aRb. Any binary relation $R \subseteq A \times B$ has an inverse $R^{-1} \subseteq B \times A$ such that $bR^{-1}a$ if-and-only-if aRb.

Appendix 1.4 Functions

(1) Let A and B be sets. A *function* or mapping from A to B is a binary relation f between A and B with the following special property: for each element $a \in A$ there is exactly one element $b \in B$ such that afb. Usually functions use prefix notation for function application writing $f(a) = b$ instead of afb. For some functions postfix notation is used to write $af = b$. To indicate that f is a function from A to B write $f : A \to B$. The set A is called the *domain* of f and the set B is called the *range* or *co-domain* of f.

(2) Consider a function $f : A \to B$ and some set $X \subseteq A$. The *restriction* of f to X is denoted by $f[X]$ and defined as the intersection of f (which is a subset of $A \times B$) with $X \times B$: $f[X] \equiv f \cap (X \times B)$. Functions may have special properties. A function $f : A \to B$ is called *one-to-one* or *injective* if $f(a_1) \neq f(a_2)$ for any two distinct elements $a_1, a_2 \in A$. It is called *onto* or *subjective* if for every element $b \in B$ there exists an element $a \in A$ with $f(a) = b$. It is called *bijective* if it is both injective and subjective.

(3) Consider an n-ary function whose domain is a Cartesian product, $f : A_1 \times \ldots \times A_n \to B$. It is customary to drop tuple brackets when applying f to a tuple $\langle a_1, \ldots, a_n\rangle \in A_1 \times \ldots \times A_n$ writing $f(2, 3)$ instead of $f(\langle 2, 3\rangle)$.

(4) Consider a binary function whose domain and co-domain coincide, $f : A \to A$. An element $a \in A$ is called a *fixed point* of f if $f(a) = a$.

(5) Boolean logic can also be considered to be functions on {*true*, *false*}.

conjunction of a and b is $a \wedge b$;

disjunction is $a \vee b$;

implication is $a \to b$;

if-and-only-if is $a \leftrightarrow b$;

exclusive disjunction is $a \oplus b$;

complement is $\neg a$.

Table 1.1: Boolean Function Truth Table

a	b	$a \wedge b$	$a \vee b$	$a \rightarrow b$	$a \leftrightarrow b$	$a \oplus b$	$\neg a$
false	*false*	*false*	*false*	*true*	*true*	*false*	*true*
true	*false*	*false*	*true*	*false*	*false*	*true*	*false*
false	*true*	*false*	*true*	*true*	*false*	*true*	*true*
true	*true*	*true*	*true*	*true*	*true*	*false*	*false*

Appendix 1.5 Sequences

(1) Sequences are ordered sets. In the following, let A be a set. A *sequence* of elements of A of length $n > 0$ is a function $f : \{1, \ldots, n\} \rightarrow A$. A sequence is denoted by listing its values in order $a_1, \ldots, a_n$ where $a_1 = f(1), \ \ldots \ , a_n = f(n)$. Then the k-th element of the sequence $a_1, \ldots, a_n$ is a_k when $k \in \{1, \ldots, n\}$. A finite sequence is a sequence of any length $n \geq 0$. A sequence of length 0 is called the *empty sequence* and denoted ϵ. A countably-infinite sequence of elements from A is a function $\xi : \mathbb{N}_0 \rightarrow A$. To exhibit the general form of a countably-infinite sequence ξ is written $\xi : a_0 \ a_1 \ a_2 \ \ldots$ when $a_k = \xi(k)$ for all $k \in \mathbb{N}_0$. Then k is called the *index* of element a_k.

(2) Consider now a set of relations, $R = \{R_1, R_2, \ldots, R_{n-1}\}$, on a set A. For any finite sequence of elements of A, $a_1 \ \ldots \ a_n$, such that each element is related to the next by a relation in R, $a_1 R_1 a_2, \ a_2 R_2 a_3, \ \ldots, \ a_{n-1} R_{n-1} a_n$ can be written as a finite chain, $a_1 R_1 a_2 R_2 a_3 \ldots R_{n-1} a_n$ For example, using the relations = and < over $\mathbb{Z}$, a finite chain may be written $a_0 < a_1 = a_2 < a_3 < a_4$. Similarly for infinite sequences and infinite chains.

(3) A *permutation* of a sequence has the same elements in different order. In the following, let f and g be sequences of distinct[2] elements $f : \{1, \ldots, n\} \rightarrow A$ and $g : \{1, \ldots, m\} \rightarrow A$. The sequences are permutations of each other, $f \leftrightarrows g$, when they are the same length and have the same elements $f \leftrightarrows g \ \equiv \ n = m \wedge \{f(1), f(2), \ldots, f(n)\} = \{g(1), g(2), \ldots, g(m)\}$

Appendix 1.6 Strings

(1) A set of symbols is often called an *alphabet*. A *string* over an alphabet A is a finite sequence of symbols from A. For example, $1 + 2$ is a string over the alphabet $\{1, 2, +\}$. Syntactic objects like AADL annex subclauses are strings.

(2) The *concatenation* of two strings s_1 and s_2 yields the string $s_1 s_2$ formed by first writing s_1 and then s_2. A string t is called a *substring* of a string s if there exist strings s_1 and s_2 such that $s = s_1 t s_2$. Because s_1 and s_2 may be empty, every string is a substring of itself.

[2] may not need restriction on repeated elements; that the sets are the same, repeated elements and all, may be enough; but then they have equal bags, not sets and that way madness lies

Appendix 1.7 Partial Orders

(1) A *partial order* is a pair $(A, \sqsubset)$ consisting of a set A and a irreflexive, antisymmetric, and transitive relation $\sqsubset$ on A. The reflexive partial order is denoted $\sqsubseteq$. If $x \sqsubset y$ for some $x, y \in A$, then x is called *less than* y, or y is *greater than* x. Consider an element $a \in A$ and a subset $X \subseteq A$. When $a \in X$ and $a \sqsubset x$ for all $x \in X - \{a\}$, the a is called the *least element* of X. When $x \sqsubset b$ for all $x \in X - \{b\}$, then b is called an *upper bound* of X. Upper bounds of X need not be elements of X. Let U be the set of all upper bounds of X. Then a is called the *least upper bound* of X if a is the least element of U.

(2) A partial order $(A, \sqsubset)$ is called *complete* if A contains a least element, and for every ascending chain $a_0 \sqsubset a_1 \sqsubset a_2 \cdots$ of elements from A, the set $\{a_0, a_1, a_2, \ldots\}$ has a least upper bound.

Appendix 1.8 Graphs

(1) A *graph* is a pair $\langle V, E \rangle$ where V is a finite set of vertices $\{v_1, v_2, \ldots, v_n\}$ and E is a finite set of edges where each edge is a pair of vertices in V, $\{\langle v_m, v_l \rangle, \ldots, \langle v_j, v_k \rangle\}$. All graphs considered here are *directed* in the the order of vertices within the pair describing the edge is significant. The set of edges forms a relation on the set of vertices $E \subseteq V \times V$. The transitive, irreflexive closure of E, called E^+ is especially important.

Appendix 1.9 Lattices

(1) A graph $\langle V, E \rangle$ is a *lattice* if the transitive, irreflexive closure of E, E^+, is an irreflexive partial order $\sqsubset$, it has a least element $\ell \in V$, and an upper bound $u \in V$. Executions of BA2015 actions create lattices.

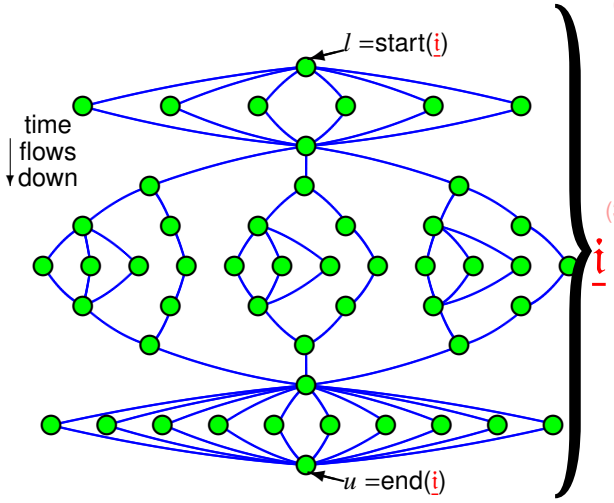

Figure 1.1: Generic Lattice

(2) Depictions of lattices place the least element (a.k.a. "start") at the top, and the greatest element (a.k.a. "end") at the bottom. Directed edges use no arrowheads; instead, the edges are presumed to flow from the higher vertex to the lower vertex.

(3) Because lattices will be used to define intervals of time, when an unspecified-further lattice needs a name it is often called $\underline{i}$. Interval $\underline{i} = \langle V_{\underline{i}}, E_{\underline{i}} \rangle$ has a least element at the top called start($\underline{i}$) $\in V_{\underline{i}}$, and an upper bound at the bottom called end($\underline{i}$) $\in V_{\underline{i}}$. Like trees and conventional current, representations are reflected; least is top (because it's first) and upper most is bottom (because it's last). To define a notion of "before" is why all that stuff about irreflexive partial orders, least elements and upper bounds was needed.

(4) Every edge and vertex in the lattice is reachable from the least element; every edge and vertex in the lattice can reach the upper bound.[3] $\forall v \in V_{\underline{i}} - \ell \mid \ell \sqsubset v \ \wedge \ \forall v \in V_{\underline{i}} - u \mid v \sqsubset u$

(5) If there is a path between v_1 and v_2, then $v_1 \sqsubset v_2$, which means v_1 occurs *before* v_2. If there is no path between v_1 and v_2, $v_1 \not\sqsubset v_2 \wedge v_2 \not\sqsubset v_1 \rightarrow v_1 \parallel v_2$ then v_1 and v_2 may occur in either order, or concurrently.

[3] $\forall$ means "for all"

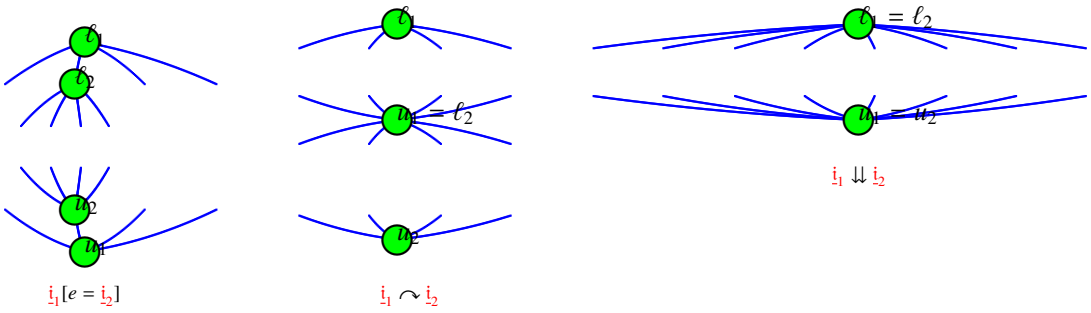

Figure 1.3: Lattice Combinations

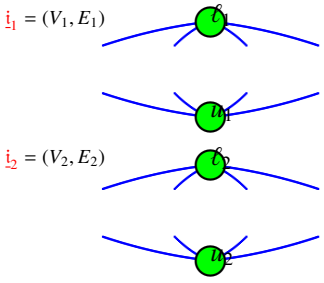

Figure 1.2: Two Lattices

(6) Lattices may combined into new lattices in three ways: sequential, concurrent, and insertion. Consider two lattices $\underline{i}_1 = \langle V_1, E_1 \rangle$ and $\underline{i}_2 = \langle V_2, E_2 \rangle$ that have no vertices in common, $V_1 \cap V_2 = \emptyset$, least elements $\ell_1 \in V_1$ and $\ell_2 \in V_2$, and upper bounds $u_1 \in V_1$ and $u_2 \in V_2$.

(7) Their *sequential lattice combination*, $\underline{i}_1 \rightsquigarrow \underline{i}_2$, may be performed as follows: substitute u_1 for ℓ_2 in V_2 and E_2, then form the union of the vertices and edges, $\underline{i}_{sc} = \langle V_1 \cup V_2, E_1 \cup E_2 \rangle$.

(8) Their *concurrent lattice combination*, $\underline{i}_1 \Downarrow\!\Downarrow \underline{i}_2$, may be performed as follows: substitute u_1 for u_2, and ℓ_1 for ℓ_2 in V_2 and E_2, then form the union of the vertices and edges, $\underline{i}_{cc} = \langle V_1 \cup V_2, E_1 \cup E_2 \rangle$.

(9) Their *insertion combination*, $\underline{i}_1[e = \underline{i}_2]$, may be performed as follows: choose an edge $e \in E_1$, $e = \langle v_j, v_k \rangle$, remove it from from E_1, substitute v_j for ℓ_2, and v_k for u_2 in V_2 and E_2, then form the union of the vertices and edges, $\underline{i}_{ic} = \langle V_1 \cup V_2, E_1 \cup E_2 \rangle$.

Appendix 1.10 Meaning

(1) The *meaning* of BA2015 language constructs is defined by giving an *interpretation* within a *context* for a *subject*:
$\mathfrak{M}_{context}[\![\text{subject}]\!] \equiv$ interpretation
where

context if given, is usually a state or set of states

subject is some construct in BA2015

interpretation is the defining formula for that subject, in that context

Appendix 1.11 Time

(1) It is important to distinguish *model time* from *real time*. Model time is a mathematical abstraction useful for defining what a system is supposed to do. Real time is where the actual systems will operate. Model time is the same everywhere in the system, and is non-negative and real, $t \in \mathbb{R} \land t \geq 0$. Real time is different everywhere; synchronous temporal domains are limited in volume by the speed of light. Great care must be used for information that flows across temporal domain boundaries. Defining such temporal domains in AADL using precise, model time is means to make them nicely work together in real time when integrated into an operational system.

(2) Periods discretize time exactly in BA2015 by choosing a countably-infinite subset of $\mathbb{R}$,

$$P_d = \{p_j \mid p_j = d\,j \text{ for all } j \in \mathbb{N}_0\}$$

where d is the period's duration, and $d\,j$ is multiplication of d by j.

(3) Defining the system's *hyperperiod* is naturally the product of all of the different durations:[4]

$$h \equiv \prod_{d \in D} d$$

where D is the set of all different durations in the system.[5]

(4) The present instant is called *now*.

(5) Many entities in BA2015 have sensible time of occurrence, T, such as events.

$$T[\![\text{e}]\!] \equiv t \mid t \in \mathbb{R} \land t \geq 0 \land \text{ e occurs at } t$$

(6) Durations are continuous sets of non-negative real numbers. Commas denote open ranges that do not include the upper and/or lower bound: ".."=closed, includes both endpoints; ",,"=open both, neither endpoint included; ",."=open left, lower bound not included; ".,"=open right, upper bound not included.

$$\begin{aligned}
\{l..u\} &\equiv \{\, m \mid m \in \mathbb{R} \land m \geq l \land m \leq u \,\} \\
\{l,,u\} &\equiv \{\, m \mid m \in \mathbb{R} \land m > l \land m < u \,\} \\
\{l,.u\} &\equiv \{\, m \mid m \in \mathbb{R} \land m < l \land m \leq u \,\} \\
\{l.,u\} &\equiv \{\, m \mid m \in \mathbb{R} \land m \geq l \land m < u \,\}
\end{aligned}$$

(7) Frequently, time will be used to define the context of meaning. Subscripted time as context notation $\mathfrak{M}_t[\![X]\!] \equiv \cdots$ is used to define the meaning for whatever X is, at a given time t. This notation is used in DZ.9.3.2 and DZ.9.4.1 to define temporal meaning for Assertions.

[4] In cases where every system-level clock is a multiple of the same reference clock, then least-common multiple of different durations can suffice.

[5] $\prod$ means "product of", usually defined over all of the numbers in a given set

Appendix 1.12 Values

(1) A *value* is a mathematical object. A *type* is a set of values (see DZ.8 Type). Values used in BA2015 are the same as the AADL Data Modeling Annex and AADL property values (which have records, but not arrays).

(2) Usually, values are singular: numbers in $\mathbb{N}_0$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, or $\mathbb{C}$; boolean in $\mathbb{B}$; a character (enclosed in apostrophes); a string (enclosed in quotation marks); or an enumeration literal (sequence of alphanumeric characters starting with a letter).

(3) More complex values are constructed from sets of pairs. Records are sets of pairs in which the first element is a record field identifier, and the second is the value of that field. Arrays are sets of pairs in which the first element is an integer index, and the second is the value of that index. Record and array values are functions in which the first element of the pair is unique. Array values generally constrain the second element of pairs to the same type. Of course, array element and record field values can be themselves be arrays or records, making arbitrarily complex values.

(4) The bottom sign, $\perp$, represents the absence of a value at a given time. The absence of value is also called *null*.

(5) The *clock operator* ^ determines when something has value. At time t,

$$\mathfrak{M}_t[\![\hat{p}]\!] \equiv \begin{array}{l} false \text{ when } \mathfrak{M}_t[\![\mathrm{p}]\!] = \perp \\ true \text{ otherwise} \end{array}$$

Appendix 1.13 States

(1) BA2015 uses two kinds of states:

lattice states variable-value bindings during actions

machine states source and destination of transitions

Appendix 1.13.1 Lattice States

(1) A *lattice state* is a set of pairs of variable names with values, with perhaps a time of the moment of occurrence.

$$L = (\{s_1, s_2, \ldots, s_m\}, t_S) \quad s_k = \langle \mathrm{n}_k, \mathrm{v}_k \rangle \quad t_S \in \mathbb{R} \wedge t_S \geq 0$$

Two states are equal if-and-only-if they have the same variables and those variables have the same values, but not necessarily the same time of occurrence. For states V and U,

$$V = (\{v_1, v_2, \ldots, v_m\}, t_V) \text{ and } U = (\{u_1, u_2, \ldots, u_m\}, t_U)$$

$$V = U \equiv \{v_1, v_2, \ldots, v_m\} = \{u_1, u_2, \ldots, u_m\}$$

Execution lattices satisfying temporal logic formulas have states as vertices (nodes).

Appendix 1.13.2 Behavior States

(1) A *behavior state* is declared in the `states` section of thread behaviors (DZ.3.2), and may be used as sources or destinations of transitions.

$$Q = (s, L)$$

Where s is a behavior state label, and L is a lattice state defining values of variables and time of occurrence.

Appendix 1.14 Arithmetic

(1) Axiomatic definitions of arithmetic have been of interest to mathematicians for centuries. For BA2015, Peano arithmetic will be assumed for natural numbers, $\mathbb{N}_0$, extended appropriately for $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$.

Appendix 1.15 Logic

(1) A *logic* is formal mathematical system for reasoning about a domain of interest. A logic consists of rules defined in terms of

symbols a set of graphical characters

formulas a set of sequences of symbols

axioms a set of distinguished formulas known to be true

rules a set of inferences to prove additional formulas from axioms, given formulas, and previously proved formulas

Not all sequences of symbols are formulas. Formulas are *well-formed* sequences of symbols. Formulas must be grammatically-correct to have meaning. A logic usually defines which sequences of symbols are formulas with a grammar.

To *satisfy* a formula means choosing values for its symbols such that the formula is true. A formula for which no choice of values for symbols is true is *unsatisfiable*. A formula always true regardless of chosen values for symbols is *tautology*.

The following formulas are assumed as axiomatic (eg. tautologies), with b, c, and d representing boolean-valued predcates, r being a bounded range, and j being an element in that range:[6]

[6] as in U.S. Pat. No. 5,867,649, col. 40, lines 40-70

Axiom 1 (*Complement*). $b \equiv \neg(\neg b)$
Axiom 2 (*Excluded Middle*). $b \vee \neg b$
Axiom 3 (*Contradiction*). $\neg(b \wedge \neg b)$
Axiom 4 (*Implication*). $a \rightarrow b \equiv \neg a \vee b$
Axiom 5 (*Equality*). $a = b \equiv a \rightarrow b \vee b \rightarrow a$
Axiom 6 (*Disjunction*). $c \vee c \equiv c$
Axiom 7 (*Disjunction*). $c \vee true \equiv true$
Axiom 8 (*Disjunction*). $c \vee false \equiv c$
Axiom 9 (*Disjunction*). $c \vee (c \wedge b) \equiv c$
Axiom 10 (*Disjunction*). $b \vee c \equiv c \vee b$
Axiom 11 (*Disjunction*). $b \rightarrow (b \vee c)$
Axiom 12 (*Conjunction*). $b \wedge b \equiv b$
Axiom 13 (*Conjunction*). $b \wedge true \equiv b$
Axiom 14 (*Conjunction*). $b \wedge false \equiv false$
Axiom 15 (*Conjunction*). $b \wedge (c \vee b) \equiv b$
Axiom 16 (*Conjunction*). $b \wedge c \equiv c \wedge b$
Axiom 17 (*Conjunction*). $(b \wedge c) \rightarrow b$
Axiom 18 (*Distribution*). $b \vee (c \wedge d) \equiv (b \vee c) \wedge (b \vee d)$
Axiom 19 (*Distribution*). $b \wedge (c \vee d) \equiv (b \wedge c) \vee (b \wedge d)$
Axiom 20 (*Universal Quantification*). $\forall j \in r \mid (b \wedge c) \equiv (\forall j \in r \mid b) \wedge (\forall j \in r \mid c)$
Axiom 21 (*Existential Quantification*). $\exists j \in r \mid (b \vee c) \equiv (\exists j \in r \mid b) \vee (\exists j \in r \mid c)$

Appendix 1.16 Computation ≡ Satisfaction

(1) BA2015 defines computation as satisfaction of interval temporal logic formulas by lattices of states.

$\mathfrak{M}_{\mathrm{i}}[\![w]\!] = true$ *(construct an interval* i *such that the formula* w *is true)*

(2) Each BA2015 program may be satisfied by a huge number of different lattices, all of which arrive at the same result.[7] The set of satisfying lattices is so large, it is effectively countably infinite. However, it suffices to consider the canonical member of the set of satisfying lattices–the shortest and bushiest lattice. Maximizing opportunities for concurrent execution is paramount for supercomputing, but embedded systems with multi-core systems-on-chip may benefit from rich opportunities for concurrent execution.

(3) For just the Action annex sublanguage, the states defined in 1.13 suffice and need no more reference in time than its position in the lattice. For satisfying lattices of states for the BA2015 annex sublanguage need time-stamps. Therefore, the set of variable-value pairs comprising a state is augmented with a real-valued time-stamp.

$$L = (\{s_1, s_2, \ldots, s_m\}, t_s) \quad s_k = \langle \mathsf{n}_k, \mathsf{v}_k \rangle$$

where t_s the time lattice state L is created. Lattice state L says nothing about the values of variables at any time other than t_s. Other lattice states could have occurred infinitesimally earlier or later. Usually, only the time-stamps of least elements and upper bounds of lattices matter, and will be ignored when they don't.

[7] Every satisfying lattice will have equal states for their least elements (start) and upper bounds (end).

Appendix 1.17 Clock

(1) A *clock* is a boolean-valued operator over machine states, S, variables, V, and ports, P which is true only when its subject has a (non-null) value: $\hat{x} \equiv \mathfrak{M}_{now}[\![x \neq \perp]\!]$. The set of all possible clock formulas, F_{SVP}, is defined with the grammar of `predicate` (DZ.9.3) augmented with the following as boolean values:

1. Ports: $\hat{p} \equiv \mathfrak{M}_{now}[\![p \neq \perp]\!]$ for port p.
2. Variables: $\hat{v} \equiv \mathfrak{M}_{now}[\![v \neq \perp]\!]$ for variable v.
3. States: $\hat{s} \equiv \mathfrak{M}_{now}[\![State(s)]\!]$ where $State(s)$ means the state machine is currently in state s.
4. Never: $c = \hat{\mathbf{0}} \equiv (\forall t = now : \neg c)$ where c is a clock formula c∈F_{SVP}.
5. Always $c = \hat{\mathbf{1}}_{SVP} \equiv (\forall t = now : c)$ where c is a clock formula c∈F_{SVP}.
6. Difference: f ^ g ≡ (f̂ **and not** ĝ) for any f,g∈F_{SVP}.
7. Next: v'=e when used in a guard of automata (D1.19) means that variable v will hold the value of expression e upon entering the destination state.[8]

(2) For example, the formula â **and** b̂ = **0̂** stipulates that ports a and b should never be assigned values simultaneously.

Appendix 1.18 Timed Formula

(1) The clock formula set F_A of an automaton A is inductively extended to the *timed formula* set $F_A^\sharp$ with atoms pertaining to real-time properties of A. Real time properties $f^\sharp \in F_A^\sharp$ are formed with the atoms $n \in \mathbb{N}_0$ and t_p, (resp. t'_p) to mean the date (number of timing periods from start) of the previous, (resp. next) occurrence of p, with integer sub-expressions $f^\sharp + g^\sharp$, $f^\sharp - g^\sharp$, for all $f^\sharp$, $g^\sharp$ in $F_A^\sharp$, and with relations $f^\sharp = g^\sharp$, $f^\sharp < g^\sharp$ for all $f^\sharp + g^\sharp$ in $F_A^\sharp$.

(2) The duration of the timing period need not be the period of the automaton (if it even has one), but is much shorter to discretize time for the system as a whole fine enough to accurately model communication in the real system.

(3) For example, the synchrony of two ports a, b is expressed as â = b̂ in F_A. In $F_A^\sharp$, it can be approximated by $d \leq t_a < d'$ and $d \leq t_b < d'$, by considering d to be the dispatch signal or date of the parent component. Literally, it means that the dates t_a and t_b of all occurrences of a and b must always occur between the dispatch date d and the next one.

[8] Not to be confused with the use of ' as a temporal operator for periodic threads meaning next period.

Appendix 1.19 Automata

(1) The behavior of a thread or device component defined with a BA2015 annex subclause is equivalent to an automation,[9] defined as a tuple:

$$A = (S_A, s_0, V_A, P_A, F_A, T_A, C_A)$$

where

S_A the set of initial, complete, execute, and final states of A

s_0 the initial state of A

V_A the set of local variables of A; $\mathbb{D}^{V_A}$ is the set of all possible values of local variables

P_A the set of ports of A, $P_A = I_A \cup O_A$, the input and output ports[10] $\mathbb{D}^{I_A}$ and $\mathbb{D}^{O_A}$ are sets of all possible values of inputs and outputs

F_{SVP} is the set of all possible clock formulas over vocabulary $W_A \equiv S_A \cup V_A \cup P_A \cup V'_A$

T_A the set of transitions $T_A \subset S_A \times \mathbb{D}^{V_A} \times \mathbb{D}^{I_A} \times F_{SVP} \rightarrow S_A \times \mathbb{D}^{V_A} \times \mathbb{D}^{O_A} \times F_{SVP}$
where $(s, V_s, I_A, g, d, V_d, O_A, f) \in T_A$

- source state $s \in S_A$,
- variable valuations $\forall v \in V_A : \mathfrak{M}_{V_s}[\![v]\!] \in \mathbb{D}$,
- input port values $\forall i \in I_A : \mathfrak{M}[\![i]\!] \in \mathbb{D}$, and
- source clock (guard) formula $g \in F_{SVP}$

map to

- the destination state $d \in S_A$,
- updated variable values $\forall v' \in V_A : \mathfrak{M}_{V_d}[\![v']\!] \in \mathbb{D}$,
- output port values $\forall o \in O_A : \mathfrak{M}[\![o]\!] \in \mathbb{D}$, and
- destination clock (finish) formula $f \in F_{SVP}$.

C_A the timing constraint $C_A \in F_{SVP}$ must equal $\hat{\mathbf{0}}$ defines timing and synchronization behavior.

(2) Let the set of all source behavior states of A (D1.13.2) and inputs be $Q_A \equiv S_A \times \mathbb{D}^{V_A} \times \mathbb{D}^{I_A}$. Equivalently, let the set of all destination behavior states and outputs of A be $Q'_A \equiv S'_A \times \mathbb{D}^{V'_A} \times \mathbb{D}^{O_A}$. Then $T_A \in Q_A \times F_{SVP} \rightarrow Q'_A \times F_{SVP}$. As shorthand, transitions may be represented as a quadruple $(s, g, d, f) \in T_A$ for *s*ource, *g*uard, *d*estination, and *f*inish, or a triple (s, g, d) where f is assumed to be true.

(3) A transition that performs action w when changing from source state s with source clock formula (guard) g to destination state d with destination clock formula (finish) f is written as $T(s, g, d, f)[w]$.

[9] denoted by a capital letter, here 'A'

[10] in out ports are members of both I_A and O_A

(4) In defining semantics with automata, a single automata may be translated into a *transition system* containing more than one transition, replacing the original transition. For $T \in T_A$:

$$T \Rightarrow T_1 \cup T_2 \equiv (T_A - T) \cup T_1 \cup T_2$$

Appendix 1.20 Synchronous Product

(1) The *synchronous product* of automata $A = (S_A, s_0, V_A, P_A, F_A, T_A, C_A)$ and $B = (S_B, t_0, V_B, P_B, F_B, T_B, C_B)$ is defined $A|B = (S_{AB}, (s_0, t_0), V_{AB}, P_{AB}, F_{AB}, T_{AB}, C_{AB})$ as follows:

$S_{AB} = S_A \times S_B$

$V_{AB} = V_A \cup V_B$

$P_{AB} = P_A \cup P_B$

$F_{AB} = F_A \vee F_B$[11]

$T_{AB} = \{((s_1, s_2), g_1 \wedge g_2, (d_1, d_2), f_1 \wedge f_2) \mid (s_1, g_1, d_1, f_1) \in T_A \wedge (s_2, g_2, d_2, f_2) \in T_B$[12]

$C_{AB} = C_A \vee C_B$

(2) Product is commutative, associative, has neutral element $(\{s\}, s, \emptyset, \emptyset, \emptyset, \emptyset, \hat{0})$ and, for deterministic automata, idempotent.

(3) The synchronous composition (immediate connection) of two automata A and B communicating through a port p is represented by the product $A|FIFOp|B$ where

$$FIFOp = \{(s_1, v' = p_{in}, s_2, true), (s_2, true, s_1, p_{out} = v)\}$$

represents the point-to-point one-place first-in-first-out behavior of port p. A port queue of size n can be specified as a series of n one-place FIFO buffers.[13]

Appendix 1.21 Small Step

(1) A *small step* is execution of a single transition of an automaton (D1.19), $T = (s, v, i, g, d, v', o, f)$. A small step leaves a source behavior state (s, v), having state label s and (persistent) variable valuation v, with the values of `in` ports i, and guard clock formula g, to enter destination behavior state (d, v'), having state label d and updated variable valuation v', sending values to `out` ports o, satisfying finish clock formula f.

[11] check with J.P.
[12] check with J.P.
[13] Wouldn't this force n steps even if the FIFO had only a single element?

Appendix 1.22 Big Step

(1) A *big step* is a finite series of small steps, such that the source state of the first transition is a and the destination of the last transition are `complete` or `pause` states.

Appendix 1.23 Trace

(1) A *port trace* is a sequence of (possibly null) values of a port. A synchronous port trace has an entry for each atom $n \in \mathbb{N}_0$ as in D1.18 with $\perp$ when the port has no value: $(p : p_0, p_1, \ldots)$. An *execution trace* of a thread is a set of traces of its ports: $\{(p : p_0, p_1, \ldots)(q : q_0, q_1, \ldots)(r : r_0, r_1, \ldots)\}$. An asynchronous trace (marked with $^\sharp$) removes all the null values.

(2) Consider execution traces $B_1 = \{(x : 2, \perp, \perp, \perp)(y : \perp, 2, 1, 0)\}$ and $B_2 = \{(x : 2, \perp, \perp)(y : 2, 1, 0)\}$. The asynchronous trace of B_1 is ${B_1}^\sharp = \{(x : 2)(y : 2, 1, 0)\}$. The asynchronous trace of B_2 is ${B_2}^\sharp = \{(x : 2)(y : 2, 1, 0)\}$. Therefore ${B_1}^\sharp = {B_2}^\sharp$.

Chapter 2

Appendix: Lexicon

*

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

Appendix 2.1 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:

[1] BA D.7(6)

```
identifier_letter
   upper_case_identifier_letter | lower_case_identifier_letter
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 2.1 defines names of certain special_characters.

Table 2.1: 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

Appendix 2.2 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 delim-

iter, 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

Appendix 2.3 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.

Appendix 2.4 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
```

Appendix 2.4.1 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.

Appendix 2.4.2 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.

Appendix 2.4.3 Rational Literals

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

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

Appendix 2.4.4 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  ]
```

Appendix 2.5 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.

Appendix 2.6 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 3

Appendix: Package and Properties

*

Two property sets and a package of data types are predeclared.

(1) Property set `BLESS` defines:

Assertion what is true about an event or data sent by, or arriving at a port

Typed data type as defined in Chapter Z.8

Invariant what is always true about a component

Precondition what must be true before a subprogram is called

Postcondition what will be true when a subprogram returns

```
property set BLESS is
  Assertion : aadlstring applies to ( all );
  Typed : aadlstring applies to ( all );
  Invariant : aadlstring applies to ( all );
  Precondition : aadlstring applies to ( subprogram );
  Postcondition : aadlstring applies to ( subprogram );
end BLESS;
```

(2) Property set `BLESS_Properties` defines:

Supported Operators what operators apply to elements of a type

Supported Relations what relations apply to elements of a type

Radix radix position for fixed-point types

```
property set \package_Properties is
  with AADL_Project;
  Supported_Operators : list of aadlstring applies to ( data );
```

```
    --used to define arithmetic operator symbols supported by a type
  Supported_Relations : list of aadlstring applies to ( data );
    --used to define relation symbols supported by a type
  Radix : AADL_Project::Size_Units applies to ( data );
    --location of the radix point for fixed-point representation
    --counting from most significant bit
end BLESS_Properties;
```

(3) These data components in the package, `BLESS_Types` represent ideal values: integers without upper or lower bounds, real numbers of infinite precision, strings with unbound length. Actual types, with ranges and bounds, must substitute for ideal types, either explicitly or automatically.

(4) Chapter Z.8 uses the standard Data Modeling annex (`Data_Model` and `Base_Types`) to define correspondence with types built in to BLESS.

```
package BLESS public
with Base_Types, BLESS_Properties, Data_Model, BLESS;
  data Integer extends Base_Types::Integer
  properties  --operators and relation symbols defined for Integer
    BLESS::Typed => "integer";
    BLESS_Properties::Supported_Operators =>
        ("+", "*", "-", "/", "mod", "rem", "**");
    BLESS_Properties::Supported_Relations => ("=", "!=", "<", "<=", ">=", ">");
  --how should conversion routines be declared?
end Integer;

data Natural extends Base_Types::Natural
  properties  --operators and relation symbols defined for Natural
    BLESS::Typed => "natural";
    BLESS_Properties::Supported_Operators =>
        ("+", "*", "-", "/", "mod", "rem", "**");
    BLESS_Properties::Supported_Relations => ("=", "!=", "<", "<=", ">=", ">");
end Natural;

data Real extends Base_Types::Float
  properties  --operators and relation symbols defined for Float
    BLESS::Typed => "real";
    BLESS_Properties::Supported_Operators => ("+", "*", "-", "/", "**");
    BLESS_Properties::Supported_Relations => ("=", "!=", "<", "<=", ">=", ">");
end Real;

data String extends Base_Types::String
  properties  --operators and relation symbols defined for String
    BLESS::Typed => "string";
    BLESS_Properties::Supported_Operators => ("+", "-"); --just concatenation
    BLESS_Properties::Supported_Relations => ("=", "!=", "<", "<=", ">=", ">");
end String;

data Fixed_Point
  properties  --operators and relation symbols defined for fixed-point arithmetic
```

```
    BLESS::Typed => "rational";
    BLESS_Properties::Supported_Operators => ("+", "*", "-", "/", "**");
    BLESS_Properties::Supported_Relations => ("=", "!=", "<", "<=", ">=", ">");
    Data_Model::Data_Representation => Integer;
end Fixed_Point;

data Time extends Real
end Time;

data flag extends Base_Types::Boolean --boolean flag
  properties
    BLESS::Typed=>"boolean";
end flag;

end BLESS_Types;
```

Chapter 4

Appendix: Alphabetized Grammar

```
action ::=
  basic_action
  | behavior_action_block
  | alternative | for_loop
  | forall_action
  | while_loop
  | do_until_loop
  | locking_action
```
§Z.6.3 p47

```
actual_assertion_parameter ::=
  formal_identifier : actual_assertion_expression
```
§Z.9.4.6 p106

```
actual_assertion_parameter_list ::=
  actual_assertion_parameter { , actual_assertion_parameter }*
```
§Z.9.3.5 p98

```
actual_parameter ::= target | expression
```
§Z.5.8 p44

```
alternative ::=
  if guarded_action { [] guarded_action }+ fi
  |
  if ( boolean_expression_or_relation ) behavior_actions
  { elsif ( boolean_expression_or_relation ) behavior_actions }*
  [ else behavior_actions ]
  end if
```
§Z.6.7 p54

```
array_range_list ::= natural_range { , natural_range }*
```
§Z.8.7 p82

```
array_size ::= [ natural_value_constant ]
```
§Z.3.3 p22

```
array_type ::= array [ array_range_list ] of type
```
§Z.8.7 p82

```
asserted_action ::=
  [ precondition_assertion ]
  action
  [ postcondition_assertion ]
```
§Z.6.2 p46

```
assertion ::=
  << ( assertion_predicate
  | assertion_function
  | assertion_enumeration
  | assertion_enumeration_invocation ) >>
```
§Z.9.2 p91

```
assertion_annex_library ::=
  annex Assertion {** { assertion }+ **} ;
```
§Z.9.1 p90

```
assertion_enumeration ::=
  asserion_enumeration_label_identifier :  parameter_identifier
  +=> enumeration_pair { , enumeration_pair }*
```
§Z.9.2.4 p93

```
assertion_enumeration_invocation ::=
  +=> assertion_enumeration_label_identifier
  ( actual_assertion_parameter )
```
§Z.9.4.7 p107

```
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
```
§Z.9.4 p102

```
assertion_function ::=
  [ label_identifier : [ formal_assertion_parameter_list ] ]
   := ( assertion_expression | conditional_assertion_function )
```
§Z.9.2.3 p93

```
assertion_function_invocation ::=
  assertion_function_identifier ( [ assertion_expression |
  actual_assertion_parameter { , actual_assertion_parameter }* ] )
```
§Z.9.4.6 p106

```
assertion_predicate ::=
  [ label_identifier : [ formal_assertion_parameter_list ] : ]
  predicate
```
§Z.9.2.2 p92

```
assertion_range ::=
  assertion_subexpression range_symbol assertion_subexpression
```
§Z.9.3.6 p99

```
assertion_subexpression ::=
  [ - | abs ]  timed_expression
  | assertion_type_conversion
```
§Z.9.4 p102

```
assertion_type_conversion ::=
  ( natural | integer | rational | real | complex | time )
  parenthesized_assertion_expression
```
§Z.9.4 p103

```
assertion_value ::=
  now | tops | timeout
  | value_constant
  | variable_name
  | assertion_function_invocation
  | port_value
```
§Z.9.4.3 p104

```
assignment ::=
  variable_name [ ' ] := ( expression | record_term | any )
```
§Z.6.4.2 p49

```
(for subprograms)
basic_action ::=
  skip | assignment | simultaneous_assignment | when_throw
  | subprogram_invocation
```
§Z.10.2 p111

```
(for threads)
basic_action ::=
  skip
  | assignment
  | simultaneous_assignment
  | communication_action
  | timed_action
  | when_throw
  | combinable_operation
  | issue_exception
  | computation_action
```
§Z.6.4 p48

```
behavior_action_block ::=
  [ quantified_variables ] { [ behavior_actions ] }
  [ timeout behavior_time ]  [ catch_clause ]
```
§Z.6.8 p56

```
behavior_actions ::=
  asserted_action
  | sequential_composition
  | concurrent_composition
```
§Z.6.1 p46

```
behavior_annex ::=
  [ assert { assertion }+ ]
  [ invariant assertion ]
  [ variables ]
  states { behavior_state }+
  [ transitions ]
```
§Z.3.1 p18

```
behavior_state ::=
  behavior_state_identifier
     : [initial] [complete] [final] state [ assertion ] ;
```
§Z.3.2 p19

```
behavior_time ::=  integer_expression unit_identifier
```
§Z.6.4.4 p50

```
behavior_transition ::=
    [ behavior_transition_label : ]
    source_state_identifier { , source_state_identifier }*
    -[ [ transition_condition ] ]-> destination_state_identifier
   [ { [ behavior_actions ] } ] [ assertion ] ;
```
§Z.3.4 p24

```
behavior_transition_label ::=
    transition_identifier [ [ priority_natural_literal ] ]
```
§Z.3.4 p24

```
behavior_variable ::=
  local_variable_declarator { , local_variable_declarator }*
  : type [ := value_constant ] [ assertion ] ;
```
§Z.3.3 p22

```
catch_clause ::=
  catch ( ( exception_label : basic_action ) )+
```
§Z.6.11 p62

```
combinable_operation ::=
 fetchadd
 ( target_variable_name ,
   arithmetic_expression [, result_identifier] )
 | ( fetchor | fetchand | fetchxor )
 ( target_variable_name , boolean_expression
   [, result_identifier] )
 | swap
 ( target_variable_name , reference_variable_name
   , result_identifier )
```
§Z.6.13 p65

```
communication_action ::=
  subprogram_invocation
  | output_port_name ! [ ( expression ) ]
  | input_port_name ? ( target )
  | frozen_input_port_name >>
```
§Z.5.1 p36

```
completion_relative_timeout_catch ::= timeout behavior_time
```
§Z.4.2 p32

```
component_element_reference ::=
  subcomponent_identifier | bound_prototype_identifier
  | feature_identifier | self
```
§Z.7.2.2 p70

```
computation_action ::=
  computation ( behavior_time [ .. behavior_time ] )
  [ in binding ( processor_unique_component_classifier_reference
    { , processor_unique_component_classifier_reference }+ ) ]
```
§Z.6.4.4 p50

```
concurrent_composition ::= asserted_action { & asserted_action }+
```
§Z.6.6 p53

```
conditional_assertion_expression ::=
  ( predicate ?? assertion_expression : assertion_expression )
```
§Z.9.4.4 p104

```
conditional_assertion_function ::=
    condition_value_pair { , condition_value_pair }*
```
§Z.9.4.5 p105

```
conditional_expression ::=
  ( boolean_expression_or_relation ?? expression : expression )
```
§Z.7.6 p73

```
condition_value_pair ::=
    parenthesized_predicate ->  assertion_expression
```
§Z.9.4.5 p105

```
constant_number_range ::=
  [ [-] numeric_constant .. [-] numeric_constant  ]
```
§Z.8.6 p80

```
data_component_name ::=
  { package_identifier :: }* data_component_identifier
   [ . implementation_identifier ]
```
§Z.8.4 p79

```
declarator ::= identifier { array_size }*
```
§Z.3.3 p22

```
dispatch_condition ::=
  on dispatch [ dispatch_expression ] [ frozen frozen_ports ]
```
§Z.4.1 p30

```
dispatch_conjunction ::=
   dispatch_trigger  { and dispatch_trigger }*
```
§Z.4.1 p30

```
dispatch_expression ::=
  dispatch_conjunction  { or dispatch_conjunction }*
  | stop
  | dispatch_relative_timeout_catch
  | completion_relative_timeout_catch
  | provides_subprogram_access_identifier
```
§Z.4.1 p30

```
dispatch_relative_timeout_catch ::= timeout
```
§Z.4.2 p32

```
dispatch_trigger ::= in_event_port_name | in_event_data_port_name
  | port_event_timeout_catch
```
§Z.4.1 p30

```
do_until_loop ::=
  do
  [ invariant assertion ]
  [ bound integer_expression ]
  behavior_actions
  until ( boolean_expression_or_relation )
```
§Z.6.10.3 p62

```
enumeration_pair ::= enumeration_literal_identifier -> predicate
```
§Z.9.2.4 p93

```
enumeration_type ::=
  enumeration
  ( defining_enumeration_literal_identifier
  { , defining_enumeration_literal_identifier }* )
```
§Z.8.5 p79

```
event ::= < port_variable_or_state_identifier >
```
§Z.9.3.10 p101

```
event_expression ::=
  [not] event
  | event_subexpression (and event_subexpression)+
  | event_subexpression (or event_subexpression)+
  | event - event                                                    §Z.9.3.10 p101

event_subexpression ::=
  [ always | never ] ( event_expression ) | event                   §Z.9.3.10 p101

event_trigger ::=
  in_event_port_component_reference
  | in_event_data_port_component_reference
  | ( trigger_logical_expression )                                   §Z.3.7 p27

exception_label ::= ( exception_identifier )+ | all                  §Z.6.11 p62

execute_condition ::=
  boolean_expression_or_relation | timeout | otherwise              §Z.3.5 p26

existential_quantification ::=
   exists logic_variables logic_variable_domain
   that predicate                                                    §Z.9.3.9 p100

expression ::=
  subexpression
  [ { + numeric_subexpression }+
  | { * numeric_subexpression }+
  |   - numeric_subexpression
  |   / numeric_subexpression
  |   mod natural_subexpression
  |   rem integer_subexpression
  |   ** numeric_subexpression
  | { and boolean_subexpression }+
  | { or boolean_subexpression }+
  | { xor boolean_subexpression }+
  |   and then boolean_subexpression
  |   or else boolean_subexpression  ]                               §Z.7.4 p72

expression_or_relation ::=
  subexpression [ relation_symbol subexpression ]                    §Z.7.5 p73

for_loop ::=
  for integer_identifier
    in integer_expression .. integer_expression
  [ invariant assertion ]
  { asserted_action }                                                §Z.6.10.2 p61

forall_action ::=
  forall variable_identifier { , variable_identifier }*
    in integer_expression .. integer_expression
    behavior_action_block                                            §Z.6.9 p58

formal_assertion_parameter ::= parameter_identifier [ ~ type_name ]  §Z.9.2.1 p91
```

```
formal_assertion_parameter_list ::=
  formal_assertion_parameter { (,) formal_assertion_parameter }*

formal_expression_pair ::=
  formal_identifier => actual_expression

frozen_ports ::= in_port_name { , in_port_name }*

function_call ::=
  { package_identifier :: }*
  function_identifier ( [ function_parameters ] )

function_parameters ::=
  formal_expression_pair { , formal_expression_pair }*

guarded_action ::=
  ( boolean_expression_or_relation )~> behavior_actions

index_expression_or_range ::=
    integer_expression [ ..  integer_expression ]

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

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

issue_exception ::=
  exception
  ( [ exception_state_identifier , ] message_string_literal )

locking_action ::=
  *!< | *!>
  | required_data_access_name !<
  | required_data_access_name !>

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

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

logical_operator ::=
  and | or | xor | and then | or else

mode_condition ::=  on trigger_logical_expression

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

§Z.9.2.1 p91
§Z.7.7 p74
§Z.4.1 p30
§Z.7.7 p74
§Z.7.7 p74
§Z.6.7 p54
§Z.7.3 p70
§Z.9.3.4 p97
§Z.3.6 p27
§Z.6.4.5 p51
§Z.6.12 p64
§Z.9.3.8 p100
§Z.9.3.8 p100
§Z.3.7 p28
§Z.3.7 p27
§Z.7.3 p70

```
natural_number ::=
 natural_integer_literal
 | natural_constant_identifier
 | natural_property
```
§Z.8.7 p82

```
natural_range ::= natural_number [ .. natural_number ]
```
§Z.8.7 p82

```
number_type ::=
  ( natural | integer | rational | real | complex | time )
  [ constant_number_range ]
  [ units aadl_unit_literal_identifier ]
```
§Z.8.6 p80

```
numeric_constant ::= numeric_literal | numeric_property
```
§Z.8.6 p80

```
parameter ::= [ formal_parameter_identifier : ] actual_parameter
```
§Z.5.8 p44

```
parameter_list ::= parameter { , parameter }*
```
§Z.5.8 p44

```
parenthesized_assertion_expression ::=
  ( assertion_expression )
  | conditional_assertion_expression
  | record_term
```
§Z.9.4.2 p104

```
parenthesized_predicate ::= ( predicate )
```
§Z.9.3.7 p99

```
port_name ::=
  { subcomponent_identifier . }* port_identifier
    [ [ natural_literal ] ]
```
§Z.5.1 p37

```
port_relative_timeout_catch ::=
  timeout ( port_identifier { [ or ] port_identifier }* )
    behavior_time
```
§Z.4.2 p33

```
port_value ::=
  in_port_name ( ? | 'count | 'fresh | 'updated )
```
§Z.7.8 p76

```
predicate ::=
  universal_quantification
  | existential_quantification
  | subpredicate
    [ { and subpredicate }+
    | { or subpredicate }+
    | { xor subpredicate }+
    | implies subpredicate
    | iff subpredicate
    | -> subpredicate ]
```
§Z.9.3 p94

```
predicate_invocation ::=
  assertion_identifier
   ( [ assertion_expression | actual_assertion_parameter_list ] )
```
§Z.9.3.5 p98

```
predicate_relation ::=
  assertion_subexpression relation_symbol assertion_subexpression
  | assertion_subexpression in assertion_range
  | shared_integer_name += assertion_subexpression
```
§Z.9.3.6 p99

```
property ::=
  property_constant | property_reference
```
§Z.7.2.1 p69

```
property_constant ::=
  property_set_identifier :: property_constant_identifier
```
§Z.7.2.1 p69

```
property_field ::=
  [ integer_value ]
  | . field_identifier
  | . upper_bound
  | . lower_bound
```
§Z.7.2.2 p70

```
property_name ::= property_identifier { property_field }*
```
§Z.7.2.2 p70

```
property_reference ::=
  ( # [ property_set_identifier :: ]
  | component_element_reference #
  | unique_component_classifier_reference #
  | self #  )
  property_name
```
§Z.7.2.2 p69

```
quantified_variables ::= declare { behavior_variable }+
```
§Z.6.8 p57

```
range_symbol ::= .. | ,. | ., | ,,
```
§Z.9.3.6 p99

```
record_field ::= defining_field_identifier : type ;
```
§Z.8.8 p83

```
record_term ::= ( { record_value }+ )
```
§Z.6.4.2 p49

```
record_type ::= record ( { record_field }+ )
```
§Z.8.8 p83

```
record_value ::= field_identifier => value ;
```
§Z.6.4.2 p49

```
relation_symbol ::= = | < | > | <= | >= | != | <>
```
§Z.9.3.6 p99

```
sequential_composition ::=
  asserted_action { ; asserted_action }+
```
§Z.6.5 p51

```
simultaneous_assignment ::=
  ( variable_name [ ' ] { , variable_name [ ' ] }+
  :=
  ( expression | record_term | any )
    { , ( expression | record_term | any ) }+ )
```
§Z.6.4.3 p50

```
subcomponent_port_reference ::=
  subcomponent_identifier { . subcomponent_identifier }*
  . port_identifier
```
§Z.3.7 p27

```
subexpression ::=
  [ - | not  | abs ]
  ( value | ( expression_or_relation )
  | conditional_expression )
```
§Z.7.5 p73

```
subpredicate ::=
  [ not ]
  ( true | false | stop
  | predicate_relation
  | timed_predicate
  | event_expression
  | def logic_variable_identifier )
```
§Z.9.3.1 p95

```
subprogram_annex_subclause ::=
  annex Action {** subprogram_behavior **} ;
```
§Z.10 p109

```
subprogram_behavior ::=
  [ assert { assertion }+ ]
  [ pre assertion ]
  [ post assertion ]
  [ invariant assertion ]
  behavior_action_block
```
§Z.10.1 p109

```
subprogram_invocation ::=
  subprogram_name ( [parameter_list] )
```
§Z.5.8 p44

```
subprogram_name ::=
  subprogram_prototype_name
  | required_subprogram_access_name
  | subprogram_subcomponent_name
  | subprogram_unique_component_classifier_reference
  | required_data_access_name . provided_subprogram_access_name
  | local_variable_name . provided_subprogram_access_name
```
§Z.5.8 p44

```
target ::= local_variable_name | output_port_name
```
§Z.5.1 p37

```
time_expression ::=
  time_subexpression
  | time_subexpression - time_subexpression
  | time_subexpression / time_subexpression
  | time_subexpression { + time_subexpression }+
  | time_subexpression { * time_subexpression }+
```
§Z.9.3.3 p96

```
time_subexpression ::= [ - ]
  ( time_assertion_value
  | ( time_expression )
  | assertion_function_invocation )
```
§Z.9.3.3 p96

```
timed_expression ::=
    ( assertion_value
      | parenthesized_assertion_expression
      | predicate_invocation )
    [ ' | ^ integer_expression | @ time_expression ]
```
§Z.9.4.1 p103

```
timed_predicate ::=
  ( name | parenthesized_predicate | predicate_invocation )
  [ ' | @ time_expression | ^ integer_expression ]
```
§Z.9.3.2 p95

```
transition_condition ::=
  dispatch_condition
  | execute_condition
  | mode_condition
  | internal_condition
```
§Z.3.4 p24

```
transitions ::= transitions { behavior_transition }+
```
§Z.3.4 p24

```
trigger_logical_expression ::=
  event_trigger { logical_operator event_trigger }*
```
§Z.3.7 p27

```
type ::=
  type_name
  | number_type
  | enumeration_type
  | array_type
  | record_type
  | variant_type
  | boolean
  | string
```
§Z.8.3 p78

```
type_name ::=
  { package_identifier :: }* data_component_identifier
    [ . implementation_identifier ]
  | natural | integer | rational | real
  | complex | time | string
```
§Z.9.2.1 p92

```
unique_component_classifier_reference ::=
  { package_identifier :: }* component_type_identifier
  [ . component_implementation_identifier ]
```
§Z.7.2.2 p70

```
universal_quantification ::=
  all logic_variables logic_variable_domain
  are predicate
```
§Z.9.3.8 p100

(for subprograms)

```
value ::=
  variable_name | value_constant | function_call
  | incoming_subprogram_parameter_identifier | null
```
§Z.10.3 p111

(for threads)

```
value ::=
  now | tops | timeout | null |
  | value_constant | in mode ( { mode_identifier }+ )
  | variable_name | function_call | port_value
```
§Z.7.1 p68

```
value_constant ::=
  true | false | numeric_literal | string_literal
  | property_constant | property_reference
```
§Z.7.2 p69

```
variables ::= variables { behavior_variable }+
```
§Z.3.3 p22

```
variant_type ::=
  variant [ discriminant_identifier ]
  ( { record_field }+ )
```
§Z.8.9 p84

```
when_throw ::=
  when ( boolean_expression ) throw exception_identifier
```
§Z.6.11 p63

```
while_loop ::=
  while ( boolean_expression_or_relation )
  [ invariant assertion ]
  [ bound integer_expression ]
  behavior_action_block
```
§Z.6.10.1 p60

Alphabetized Lexicon

```
base ::= digit [ digit ]
```
§2.4.2 p131

```
based_integer_literal ::=
  base # based_numeral # [ positive_exponent ]
```
§2.4.2 p131

```
based_numeral ::= extended_digit [_] extended_digit
```
§2.4.2 p131

```
character ::= graphic_character | format_effector
    | other_control_character
```
§2.1 p128

```
comment ::= --{non_end_of_line_character}*
```
§2.6 p132

```
complex_literal ::=
  [  [-] real_literal :  [-] imaginary_part_real_literal  ]
```
§2.4.4 p132

```
decimal_integer_literal ::= numeral
```
§2.4.1 p131

```
decimal_real_literal ::= numeral . numeral [ exponent ]
```
§2.4.1 p131

```
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```
§2.1 p129

```
exponent ::= (E|e) [+] numeral | (E|e) - numeral
```
§2.4.1 p131

```
extended_digit ::=
  digit | A | B | C | D | E | F | a | b | c | d | e | f
```
§2.4.2 p131

```
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).
```
§2.1 p129

```
graphic_character ::= identifier_letter | digit | space_character
     | special_character
```
§2.1 p128

```
identifier ::= identifier_letter {[_] letter_or_digit}*
```
§2.3 p130

```
identifier_letter
   upper_case_identifier_letter | lower_case_identifier_letter
```
§2.1 p128

```
integer_literal ::= decimal_integer_literal | based_integer_literal
```
§2.4 p131

```
letter_or_digit ::= identifier_letter | digit
```
§2.3 p130

lower_case_identifier_letter
  Any character of Row 00 of ISO 10646 BMP whose name begins
  Latin Small Letter. §2.1 p129

numeral ::= digit {[_] digit}* §?? p??

numeric_literal ::=
  integer_literal | real_literal | rational_literal | complex_literal §2.4 p131

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. §2.1 p129

rational_literal ::=
  [[−] *dividend*_integer_literal | [−] *divisor*_integer_literal] §2.4.3 p132

real_literal ::= decimal_real_literal §2.4 p131

space_character
  The character of ISO 10646 BMP named Space. §2.1 p129

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. §2.1 p129

string_element ::= "" | non_string_bracket_graphic_character §2.5 p132

string_literal ::= "{string_element}*" §2.5 p132

upper_case_identifier_letter
   Any character of Row 00 of ISO 10646 BMP whose name begins
  Latin Capital Letter. §2.1 p129

Index