Annex Document D Behavior Model

Normative

Annex D.1 Scope

(1) The Behavior Annex document provides a standard sublanguage extension to allow behavior specifications to be attached to AADL components. The aim of the Behavior Annex is to refine the implicit behavior specifications that are specified by the core of the language. The Behavior Annex targets the following goals:

- Describe the internal behavior of component implementations as a state transition system with guards and actions. However, the aim is not to replace software programming languages or to express complex subprogram computations.
- Extend the default run-time execution semantics that is specified by the core of the standard, such as thread dispatch protocols.
- Provide more precise subprogram calls synchronization protocols for client-server architectures.

(2) The Behavior Annex document includes the definition of the Behavior_Specification AADL annex subclause that is used to describe internal behavior and run-time semantics extensions and the Behavior_Properties property set that defines a property for subprogram call protocols. This document is organized as follows:

- Annex D.2 gives a short overview of the annex
- Annex D.3 defines the syntax and semantics of the **state automaton** used for the Behavior_Specification annex subclause.
- Annex D.4 defines the syntax and semantics of the **dispatch condition language** used to specify the conditions for a thread dispatch that are a refinement of the default thread dispatch conditions specified in the core AADL standard.
- Annex D.5 defines the syntax and semantics of the **interaction operations** used to specify the component interaction with other components through its port, subprogram, and data access features.
- Annex D.6 defines the syntax and semantics of the **action language** used to specify the transition actions of the automaton.
- Annex D.7 defines the syntax and semantics of the **expression language** used in the various parts of the behavior annex.
- Annex D.8 provides details about the behavior of subprogram call **synchronization protocols** defined by the Subprogram_Call_Protocol property.

Annex D.2 Overview of Behavior Annex Concepts

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

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

(3) These state transition systems can be used to specify the sequential execution behavior of an AADL subprogram, the dispatch, mode, input, and output behavior of AADL threads or devices, the protocol behavior of AADL virtual buses and buses, the dynamic behavior of a process or system, etc. The behavior from the initial state to a final state typically represents the execution behavior of a subprogram with one or more return points. The behavior of a component such as a sampling periodic thread or a thread processing commands, may start in an initial state; initialize itself and suspend itself at a complete state; reactivate from the complete state repeatedly based on time or the arrival of an external event or message; transitioning from the initial state to the first complete state, or between complete states may involve transitioning to intermediate computational states; a termination request results in a transition to a final state.

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

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

(6) When a component is specified in the core AADL model to have modes, then the Behavior_Specification supports the specification of mode-specific behavior.

(7) A subprogram call is an external trigger to the state transition system of a subprogram. The arrival of events and event data on ports of 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 [AS5506A 5.4.1] as well as the state transition system of the Behavior_Specification of the thread. The transmission request on an outgoing port is an external trigger to the state transition system of a virtual bus or bus. Dispatch conditions are specified in terms of external triggers via event port, event data port, calls received on provides subprogram access features, or time out. Dispatch does not depend on the input value, which can only impact the action following the dispatch.

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

(9) Input on ports is frozen according to the semantics of the core AADL standard [AS5506A 8.3.2] and made available to the application Behavior_Specification in the form of a port variable. Newly arriving data, events, and event data do not affect the content of the port variable. In the case of a data port the current value at input freeze time is made available. In the case of event ports and event data ports the port queue content is handled according to the specified dequeuing protocol. One, several, or all dequeued elements are made available to the Behavior_Specification.

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

Annex D.3 Behavior Specification

(1) Behavior specifications can be attached to any AADL component types and component implementations using an annex subclause. When defined within component type specifications, it represents behavior common to all the associated implementations. If a component type or implementation is extended, behavior annex subclause defined in the ancestor are applied to the descendent except if the later defines its own behavior annex subclause.

(2) The Behavior Annex supports behavior specification for components with modes in two ways. First, a behavior annex subclause can be specified for a specific mode by declaring it with an **in modes** statement [AS5506A 12]. If mode specific behavior annex subclauses are defined, those modes for which no behavior annex is given have an empty behavior specification. If no mode specific behavior annex subclause is defined, then it applies for all modes.

(3) Second, modes can be reflected in a Behavior_Specification without **in modes**. In this case a mode can be reflected by a complete state of the same name in the Behavior_Specification and mode transition behavior can be modeled as a transition out of such a complete state whose condition identifies the event port named in the mode transition, if specified in the core AADL model.

(4) The behavior annex defines a transition system (an extended automaton) described by three sections:
- variables declarations;
- states declarations;
- transitions declarations;

(5) The variables section declares identifiers that represent variables with the scope of the behavior annex subclause in which they are declared.

(6) Local variables can be used to keep track of intermediate results within the scope of the annex subclause. They may hold the values of out parameters on subprogram calls to be made available as parameter values to other calls, as output through enclosing out parameters and ports, or as value to be written to a data component in the AADL specification. They can also be used to hold input from incoming port queues or values read from data components in the AADL specification. They are not persistent across the various invocations of the same behavior annex subclause.

(7) State variables can be used to reduce the size of the state automaton by keeping track of counts. Such state variables must be declared in the core specification as data subcomponents or requires data access features.The same visibility rules as those defined in the core standard apply to the behavior annex subclause, for instance, a behavior annex subclause defined in a component type declaration cannot reference a data subcomponent as a state variable.

(8) The states section declares all the states of the automaton. Some states may be qualified as *initial* state, *final* state, or *complete* state, or combinations thereof. A state without qualification will be referred to as execution state. A behavior automaton starts from an initial state and terminates in a final state.

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

(10) In the case of threads and devices, the automaton consists of one *initial* state representing the state before initialization actions, one or more *complete* states, zero or more intermediate execution states, and one or several *final* states. A *complete* state acts as a suspend/resume state out of which threads and devices are dispatched. The *final* state represents the state when a thread or device completes finalization. In the case of threads, these qualified states have equivalents in the hybrid automata that define the runtime system behavior in the core AADL standard [AS5506A 5.4.1]. Note that one state can pay the role of *initial*, *complete*, and *final* state.

(11) Other components, such as processes, have behavior automata with an *initial* state, one or more *complete* states and one *final* state. The *initial* state represents the state before initialization begins. The *final* state represents the state after finalization completes.

(12) A state that is qualified as *final,* and is not at the same time *initial* or *complete*, cannot accept outgoing transitions. If the purpose of the behavior annex is to provide a specification of the intended behavior of a component, then the use of several *final* states is allowed. Otherwise, if the purpose is to provide a deterministic representation of the implementation of the internal behavior of the component, then only one *final* state must be defined.

(13) When a subprogram has modes it may have a separate behavior annex subclause for each mode. In this case, a mode transition results in a transition to the *initial* state of the behavior automaton of the new mode at the next subprogram call.

(14) When a component other than a subprogram has modes it may also have a separate behavior annex subclause for each mode. In this case, a mode transition results in a transition from the *complete* state of the current mode behavior automaton to the *initial* state of the behavior automaton of the new mode.

(15) The transitions section defines transitions from a source state to a destination state. Transitions in a subprogram behavior automaton represent the control flow sequence through execution states to a *final* state. Transitions in a thread behavior automaton represent the execution sequence within a thread. A transition out of a *complete* state is initiated by a dispatch once the dispatch condition is satisfied.

(16) Transitions can be guarded by dispatch conditions, or execute conditions.

(17) Dispatch conditions explicitly specify dispatch trigger conditions out of a *complete* state. If not specified, the default dispatch conditions provided by the core AADL standard apply. A dispatch condition is a Boolean expression that specifies the logical combination of triggering events for the next dispatch. A dispatch triggering event can be the arrival of an event or event data on an event port or an event data port, the receipt of a call on a provided subprogram access, or a time out event. The ports listed in the dispatch condition must be consistent with the ports listed in the core AADL model as dispatch triggers.

(18) Execute conditions specify transition conditions out of an execution state to another state. They can also be used to express initialization sequence when applied on a transition out of a state that is qualified as *initial* and is not at the same time *complete* or *final*. They effectively select between multiple transitions out of a given state to different states. These conditions are logical expressions based on component inputs, subcomponent outputs, and values of data components, state variable values, and property constants. They can also result in catching a previously raised execution Timeout exception.

(19) If transitions have been assigned a priority number, then the priority determines the transition to be taken. The higher the priority number is the higher the priority of the transition is. If more than one transition out of a state evaluates its condition to true and no priority is specified, then one transition is chosen non-deterministically. For multiple transitions with the same priority value the selection is also non-deterministic. Transitions with no specified priority have the lowest priority.

(20) Each transition can have actions. Actions can be subprogram calls, retrieval of input and sending of output, assignments to variables, read/write to data components, and time consuming activities. An action is related to the

transition and not to the states: if a transition is taken, the sequence of actions is performed and then the state specified as the destination of the transition becomes the new current state.

Synchronous Behaviors

In a behavior specification, a transition marks the consumption of time perceived by the scheduler from the time its defining thread is dispatched (its source state) to the time it releases control (target state). A delayed transition can start or end from or to an execution state.

Accordingly, in a synchronous specification, complete states temporally mark the interaction of a behavior with its parent scheduler (static or dynamic) and implicitly define time boundaries (called instants). But this is not enough to model concurrent synchronous behaviors (in the case of static scheduling). To statically schedule (immediate or delayed) communications between synchronous behaviors, one must ensure that two consecutive events along the same port (immediate or delayed) be separated by either a complete state, to release control to the scheduler, or by a **`pause`** event, to synchronize with concurrent behaviors by locally suspending/resuming execution between two (execution) steps. A **`pause`** event, or a **`pause`** action statement, allows to explicitly mark this logical time boundary.

Semantics

(21) A behavior annex subclause is used to specify behaviors of components, such as the interaction behavior with other components, discrete state behavior of the component itself including time consuming actions, functional behavior based on external conditions, protocol behavior of virtual buses and buses, logical control flow in terms of subprogram calls within a thread or a subprogram. These behavior specifications must be consistent with the respective semantics in the core AADL standard.

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

(23) The core AADL standard defines runtime execution states for threads. These states include an *initial* state (thread halted), a *complete* state (awaiting dispatch,) and a *final* state (stopped thread). The behavior annex specification for a thread must be consistent with the core AADL semantics.

(24) The behavior specification of components other than subprograms consists of an *initial* state, one or more *final* states, one or more *complete* states, and zero or more execution states. A transition out of the *initial* state is triggered by the *initialize* action defined in the core AADL standard. Execution states may be used to represent intermediate initialization steps. Upon completion of initialization a *complete* state is reached. 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.

(25) A dispatch trigger will result in a transition out of a complete state and to one of the states defined in the behavior annex (either an execution state or a complete state). A dispatch trigger will result in a transition out of a complete state to zero or one execution state. A dispatch trigger can be the arrival of input on ports, a subprogram call initiated by another thread, or a timed event (periodic dispatch or timeout). Reaching a *complete* state can be interpreted as calling the `Await_Dispatch` run-time service. Thus a component is suspended if it performs a transition to a *complete* state, after having executed the action associated to the transition. The next dispatch will restart the thread from that state.

(26) 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.

(27) The dispatch conditions are evaluated to determine whether a dispatch occurs. If there are multiple outgoing transitions the execution condition (if present) is evaluated to determine which transition is taken. If multiple transitions are eligible then the priority value determines an evaluation ordering, otherwise one of the eligible transitions is taken non-deterministically. The higher the priority value is, the higher the priority of the transition is.

(28) 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.

(29) Local variables are temporary placeholders for application data that may not require temporary storage. For example, they are used to indicate which out parameter of a subprogram call is used as in parameter of another call or as output in a port variable. Local variables must be explicitly typed with a valid data component classifier.

(30) An *otherwise* execute condition becomes true when all the other execute conditions associated with transitions from the same state are false.

Synchronous Semantics

As a result of the above, the transition system of a (synchronous) behavior annex is equivalent to an automaton A=(S,X,T) in the form of [SCP'14, page 4[1]] where:

- S is its finite set of the initial, execution, complete and final states S°, S^{C}, S^{E} and S^{F}
- X is its finite set of variables V and immediate and delayed input and output ports I and O
- T : S x I -> O x S is the transition system

A transition *(s,i,o,s')* of an automaton

- Evaluates the Boolean formula i : I x V -> Bool defined on its input ports and variables
- Transits from state s to state s'
- Evaluates the function o : I x V -> O x V on its variables and output ports
- starts from an initial, complete or execution state in *S* by the dispatch of a set of input ports *I* and/or a Boolean expression on its variables *V,*

In the remainder, a simple transition from *s* to *s'* that receives *a* and sends *b* is written *(s,a,b,s')* or *s-a/b->s'*

Outputs along immediate ports *O* and shared variables *V* are available immediately to the current and concurrent behaviors, and before the source behavior reaches a complete state. Outputs along delayed ports *O'* are available once the source behavior has reached a complete state.

Port communications between synchronous behaviors are synchronous. Delayed communications transit via an automaton that represents the protocol of that communication (e.g. FIFO). For instance, a one-place buffer between the source *p* and target renamed *p'* of a delayed port p can be represented by: s-p?(v)/!**pause**->s'-/p'!(v)->s by marking a logical time barrier between the emission from p to the receipt by p' using a (virtual) **pause** event.

[1] L. BESNARD , A. BOUAKAZ , T. GAUTIER , P. LE GUERNIC , Y. MA , J.-P. TALPIN , H. YU . Timed behavioural modelling and affine scheduling of embedded software architectures in the AADL using Polychrony , in "Science of Computer Programming", July 2014. https://hal.inria.fr/hal-01095010

Syntax

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

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

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

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

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

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

behavior_transition_priority ::=
  numeral

behavior_condition ::=
  dispatch_condition
| execute_condition

execute_condition ::=
  [ logical_value_expression | behavior_action_block_timeout_catch | otherwise ]

timeout_catch ::=
  timeout

declarator ::=
  identifier { array_size }*

array_size ::
  [ integer_value_constant ]
```

Naming Rules

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

(N2) An empty execute condition is equivalent to a condition that is always true.

Legality Rules

(L1) A behavior annex specification for a subprogram must define an initial state and one final state.

(L2) A behavior annex specification for a subprogram must not define any complete states.

(L3) A behavior annex specification for a thread, device, and other components that can be suspended awaiting dispatch or awaiting a mode transition, must define at least one complete state and one initial state. This may be the same state.

(L4) A behavior annex specification for threads and other components with initialization and finalization entrypoints may explicitly model the initialization and finalization by including one initial state and one or more final states.

(L5) A behavior annex specification for a subprogram must not contain a dispatch condition in any of its transitions.

(L6) Only transitions out of complete states may have dispatch conditions.

Note – Per (L6), and to maintain the equivalence between action blocks and transition systems, a dispatch condition cannot occur inside an action block, as the entry point of the action block is an implicit execution step whose source is the parent transition complete state. Consequently, the logical time laps between two successive reactions of a synchronous behavior are marked by signaling it to the synchronous scheduler (static or dynamic) using the virtual **pause** output event.

(L7) Transitions out of complete states must have dispatch conditions.

(L8) Transitions from states that are final only are not allowed.

(L9) The specification of frozen ports in the dispatch condition must be consistent with that of the core AADL model [AS5506A 5.4.8].

Consistency Rules

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

(C2) A behavior annex subclause of a subcomponent overrides the behavior annex subclause of its containing component if they conflict

(C3) The behavior annex state transition system must not remain blocked in an execution state. This means that the logical disjunction of all the execute conditions associated with the transitions out of an execution state must be true.

(C4) If the behavior annex defines transitions from a complete state that represents a mode in the containing component, then the behavior_condition associated with these transitions must be consistent with the corresponding mode_transition_triggers [AS5506A 12].

Examples

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

package BA_example1 public

data number
```

```
end number;

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

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

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

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

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

end BA_example1;

-- Next example exploits the AADL execution model to compute a speed by
-- counting the number of ticks received each second by the measuring thread.
-- The event port queue acts as a persistent count. At each periodic
-- dispatch, all events received during the previous period are transferred
-- to a thread local queue (implemented as counter) and become available to
-- the user. At that time the event port queue is reset to empty (count
-- zero). The Dequeue_Protocol of AllItems specifies that all events are to
-- be removed from the event port queue.

package BA_example2 public
with Base_Types;

thread speed
  features
    tick: in event port { Dequeue_Protocol => AllItems; };
    sp: out data port Base_Types::Integer;
  properties
```

```
    Dispatch_Protocol => periodic;
    Period => 1 sec;
end speed;

thread implementation speed.i
  annex behavior_specification {**
    states
      s0: initial complete final state;
    transitions
      s0 -[ on dispatch ]-> s0 { sp := tick'count };
  **};
end speed.i;

-- Note that in the preceding code, tick'count provides the number of events
-- that are available in the tick port queue.

end BA_example2;
```

Annex D.4 Thread Dispatch Behavior Specification

(1) 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.

(2) A dispatch condition can only be associated with a transition out of a *complete* state. In this case if the dispatch condition evaluates to true the thread dispatches and the transition is taken. If no dispatch trigger condition is specified, then the condition specified in core AADL specification applies. If no frozen ports are specified, then the ports are frozen according to the core AADL specification.

(3) A dispatch trigger condition can be the arrival of events or event data on ports (expressed as a disjunction of conjunctions), calls on provides subprogram access features, the **stop** event, and occurrence of dispatch related and completion related time outs.

(4) Periodic dispatches are always considered to be implicit unconditional dispatch triggers on complete states and handled by dispatch conditions without dispatch trigger condition.

(5) Timeout is a dispatch trigger condition 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. Completion related timeout values are specified as part of the completion_relative_timeout_condition_and_catch declaration.

Note – A transition carrying a dispatched timeout has outmost priority over these that do not.

(6) **Stop** events are dispatch triggers to model initiation of finalization and transition from a *complete* state to the *final* state, possibly via one or more execution states. If the core property *finalize entrypoint* is already specified [AS5506A 5.4.1] then it can be used as an implicit finalization action, otherwise it can be specified as BA action on transitions from complete states.

(7) In a behavior specification, a *final* state can be a complete state (i.e. an *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.

(8) In a behavior specification, 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.

Syntax

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

dispatch_trigger_condition ::=
  dispatch_trigger_logical_expression
| provides_subprogram_access_name
| provides_subprogram_access_identifier
| stop
| completion_relative_timeout_condition_and_catch
| dispatch_relative_timeout_catch

dispatch_trigger_logical_expression ::=
  dispatch_conjunction { or dispatch_conjunction }*

dispatch_conjunction ::=
  dispatch_trigger { and dispatch_trigger }*

completion_relative_timeout_condition_and_catch ::=
  timeout behavior_time

dispatch_relative_timeout_catch ::=
  timeout

dispatch_trigger ::=
  in_event_port_name
| in_event_data_port_name
dispatch_trigger ::=
  in_event_port_identifier
| in_event_data_port_identifier

frozen_ports ::=
  in_port_identifier name { , in_port_identifier name }*
```

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.

(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.

Legality Rules

(L1) The following table 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.

dispatch_protocol / dispatch_trigger_condition	*Periodic*	*Sporadic*	*Aperiodic*	*Hybrid*	*Timed*
∅	X			X	X
dispatch_trigger (logical combination)		X	X	X	X
Provides_subprogram_access		X	X	X	X
Stop	X	X	X	X	X
dispatch_relative_timeout_catch					X

(L2) The dispatch_relative_timeout_catch statement must only be declared for timed threads, and must be declared in only one outgoing transition of a complete state.

(L3) The completion_relative_timeout_condition_and_catch statement must be declared in at most one outgoing transition of a complete state. Besides, note that the completion_relative_timeout_condition_and_catch statement is not present in the previous table because this statement cannot be applied to a thread component (but only to device components). Indeed, the completion_relative_timeout_condition_and_catch is not compatible with any of the possible values of the dispatch_protocol property of a thread.

Examples

```
-- Next example shows various dispatch conditions and transition actions. The
-- timeout value is given by the Period property of the thread. States st and
-- sf are complete states and thus outgoing transitions are triggered by
-- dispatch conditions, whereas s1 and s2 are execute states and outgoing
-- transitions are guarded by logical expressions.

package BA_example3 public
with Base_Types;

thread sender
features
  d: out event data port Base_Types::Integer;
  a: in event data port Base_Types::Integer;
properties
  Dispatch_Protocol => Timed;
  Period => 10 ms;
end sender;
```

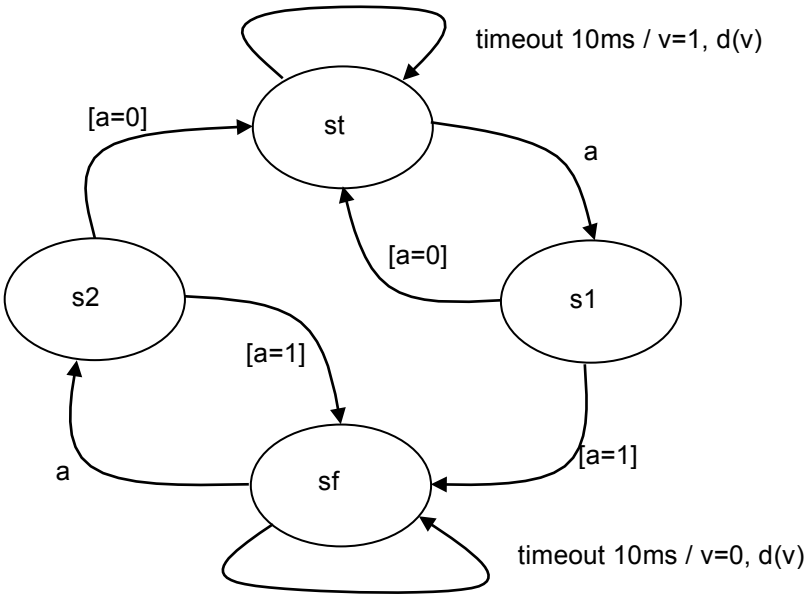

Figure 1: Sender Behavior Specification

```
thread implementation sender.v1
subcomponents
  v: data Base_Types::Integer;
annex behavior_specification {**
  states
    st: initial complete state;
    sf: complete final state;
    s1, s2: state;
  transitions
    st -[on dispatch timeout]-> st { v := 1; d!(v) };
    st -[on dispatch a ]-> s1;
    s1 -[a=1]-> sf;
    s1 -[a=0]-> st;
    sf -[on dispatch timeout]-> sf { v := 0; d!(v) };
    sf -[on dispatch a ]-> s2;
    s2 -[a=0]-> st;
    s2 -[a=1]-> sf;
**};
end sender.v1;

-- In a second version, it is possible to eliminate the need to maintain
-- state in the data component v. The state information is already reflected
-- in the state automaton.

thread implementation sender.v2
annex behavior_specification {**
  states
    st: initial complete state;
    sf: complete final state;
    s1, s2: state;
  transitions
    st -[on dispatch timeout]-> st { d!(1) };
    st -[on dispatch a ]-> s1;
    s1 -[a=1]-> sf;
    s1 -[a=0]-> st;
    sf -[on dispatch timeout]-> sf { d!(0) };
```

```
    sf -[on dispatch a ]-> s2;
    s2 -[a=0]-> st;
    s2 -[a=1]-> sf;
**};
end sender.v2;

end BA_example3;
```

Annex D.5 Component Interaction Behavior Specification

(1) Threads can interact through shared data, 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.

(2) Messages can be received by the annex subclause 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.

(3) 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 [AS5506A 9.2.4] and initiated by a `Receive_Input` service call [AS5506A 8.3.5] 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.

(4) 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 [AS5506A 8.3.5] and in the Behavior_Specifcation via functions (see paragraph (8) below).

(5) Event and event data ports have queues and the queues are processed as follows according to the value of the `Dequeue_Protocol` defined in the core standard [AS5506A 8.3.3].

- The *OneItem* `Dequeue_Protocol` removes one item from the port queue and makes it available through a port variable with the name of the port. If the queue is empty the port variable content is considered not fresh.
- The *AllItems* `Dequeue_Protocol` removes all items from the port queue and makes them available through a port variable with the name of the port. The values are kept in a port variable internal queue and the size of this internal queue can be determined by the `p'count` operation (see below). In the Behavior_Specification an element can be dequeued by the `p?` operation and become accessible through the port variable.. Any data not processed from the port variable internal queue is lost at the next time input is frozen.
- The *MultipleItems* `Dequeue_Protocol` freezes the content of the port queue and makes it available to the thread. In this case the thread can remove elements from the port queue up to the number of frozen elements. The number of elements available for dequeuing can be determined by the `p'count` operation (see below). In the Behavior_Specification an element can be dequeued by the `p?` operation and become accessible through the port variable. Any elements not consumed remain in the port queue and are available at the next execution together with any newly arrived elements.

(6) 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.

Similarly, initiating transmission of port output must be consistent between the `Output_Time` property in the core model and the port output (`p!`) action in the Behavior_Specification.

(7) Ports causing a dispatch event are implicitly frozen at dispatch time. It is also possible to explicitly freeze additional ports. If the `Input_Time` property of a port has the value *Dispatch_Time*, then an explicit call to the `Receive_Input` service can be performed thanks to the **with** statement of the dispatch condition. If the `Input_Time` property of a port has the value *Start* or *Completion*, then an explicit call to the `Receive_Input` service can be performed thanks to the input freeze operator (**>>**) as a transition action.

(8) At the time the input is frozen, the content of a data port, event port, or event data port is frozen and it is available to the application within a thread through a port variable. In the case of a data port it is the most recent data value, while in the case of an event or event data port the behavior depends on the value of the `Dequeue_Protocol` property:

- Under the *OneItem* `Dequeue_Protocol` the first element of the port queue is made available through the port variable name.
- Under the *AllItems* `Dequeue_Protocol` all elements of the port queue are made available through a port variable internal queue with the first element directly accessible via the port variable name.
- Under the *MultipleItems* `Dequeue_Protocol`, elements of the port queue up to a specified number can be explicitly dequeued one by one and are accessible via the port variable name.

(9) Within a Behavior_Specification, the following constructs are available to get the status and the contents of a port `p`:

- `p` returns the data value stored in the frozen port variable. Multiple reference to *p* return the same value unless a dequeue or an input freeze is performed. The returned value may not be fresh if no new value arrived at the port at input freeze or the port queue is empty.
- `p'count` returns the number of elements available through the frozen 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. This number decreases every time an element is dequeued.
- `p'fresh` returns true if the frozen port variable contains a new value. Otherwise it returns false. 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**.
- `p?` dequeues an event or event data from a frozen and non empty event port queue and in the case of an event data port returns its data value. The value of `p'count` is decremented. In the case of an event data port the new first element is available in the port variable. If the resulting queue is empty the port variable value remains the previous one and is considered not to be **fresh**.
- `p?(x)` dequeues an element from the local queue of a frozen event data port and assigns it to the local variable x. The value of p**'count** is decremented.

(10) Within the behavior annex 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.

(11) 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.

(12) Access to shared data subcomponents is controlled according to the `Concurrency_Control_Protocol` property specified associated with this data subcomponent [AS5506A 5.1]. 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) operators within a behavior actions 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 [AS5506A 8.6] 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 [AS5506A 5.1.1] that can be achieved using the **!<** (resp. **!>**) operators applied to the shared data subcomponent identifier.

(13) Output can be sent within the annex subclause through declared features of the containing component type. These can be out or in out data ports; out or in out event ports; out or in out event data ports, out or in out parameters of the containing subprogram, and write access to data subcomponents directly or through data access features. Output can also be sent as incoming parameter values to subprogram calls.

(14) The generation of output is consistent with the timing semantics of the core language. The core language specifies the output time on ports and data access features through a `Output_Time` property and the sending of the output is initiated by a `Send_Output` service call [AS5506A 8.3.2] 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).

(15) Within the behavior annex, the following constructs are available to set the contents of a port `p`:

- `p!` calls `Send_Output` on an event port, data port, or event data port. Transmission of the event or data is initiated immediately - consistent with the Output_Time property. The value of the port variable is sent in the case of data ports or event data ports. The port variable value may have been assigned within the same behavior action sequence, may be the initial port variable value, or may be a previously sent value.

- `p := v` assigns the data value *v* to the port variable of a data or event data port *p*. Data is transferred to the destination port at the next `p!` call or after deadline or completion for data ports and event data ports that have not been sent explicitly.

- `p!(v)` assigns the data value `v` to the port variable of a data port or event data port `p` and calls the `Send_Output` service. Transmission of the event or data is initiated immediately - consistent with the `Output_Time` property.

Note – If a port connection is immediate, the data and/or event sent from the source of the connection (the sender) must be dispatched to the target of the connection (the recipient) before the next dispatch trigger of the sender.

(16) 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`.

(17) 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.

(18) 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, port values or assigned temporary variables.

(19) Within the behavior annex, the following constructs are available to call a subprogram `s`, where `s` may be the name of either a subprogram prototype, a requires subprogram access feature or a provides subprogram access feature of a requires data access feature of the component type, a subprogram subcomponent of the component implementation or a subprogram classifier in a visible package:

- `s!` calls the parameterless subprogram referred to by `s`.
- `s!(par1,…,parn)` calls the subprogram referred to by `s` with the corresponding actual parameter list, i.e., value expressions for incoming parameters, and locally visible variables to hold the values of outgoing parameters. The call may include references to shared data components or data access features to model reference parameters. Ordering of actual parameters must be the same as the one of the parameter feature declarations in the corresponding subprogram type.

(20) Provides subprogram access features that are declared in the 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.

(21) 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 [AS5506A 5.2]. To specify non-blocking calls, a *SemiSynchronous* `Subprogram_Call_Type` property must be applied to the subprogram.

Consistency Rules

(C1) If the freezing of an input port is specified with the Input_Time property [AS5506A 8.3.2], then no freeze input action must be specified in the corresponding dispatch condition (**frozen** section) or behavior actions (**>>** operator) of the behavior subclause, or the two statements must be equivalent.

(C2) If the sending time of an output port is specified with the Output_Time property [AS5506A 8.3.2], then no send output action must be specified in the corresponding behavior actions (**!** operator) of the behavior subclause, or the two statements must be equivalent.

Annex D.6 Behavior Action Language

(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.

(2) Basic actions can be assignment actions, communication actions or time consuming actions.

(3) 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 references of assignments are local variables, data components acting as persistent state variables, and outgoing features such as ports and parameters.

(4) 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.

(5) Timed actions can be pre-defined **`computation`** actions. **`Computation`** actions specify computation time intervals.

(6) An execution Timeout exception can be raised after any behavior action block. Raising such a timeout event may trigger a transition with a timeout catch execute condition.

Semantics

(7) Behavior action blocks are associated with transitions and are performed when the transition is taken. Behavior action blocks can be sequences of actions that are performed in order (character **;** as separator) or sets of actions that are performed independently (character **&** as separator). Action sets can be elements of action sequences and vice versa. As such they define partially ordered multi-sets.

(8) Control constructs support conditional execution of alternative actions (**if** construct), conditional repetition of actions (**while** and **do .. until** constructs), and application of actions over all elements of a data component array, port queue content, or integer range (**for** and **forall** construct). The **for** construct represents an ordered iteration over all elements, while the **forall** construct applies the actions to all elements in any order. Within the **for** or **forall** constructs the element can be referenced by the *element_identifier*, which acts as a local variable with the name scope of the **for** or **forall** construct.

Note – the successive outputs to a port in the body of a ***while*** *or* ***for*** *loop must be marked by a logical time barrier* ***pause****.*

(9) Actions in a behavior action block can be basic actions in the form of communication actions, assignment actions, and timed actions, or they can be action sets or action sequences.

(10) Communication actions support interaction with other components. 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`**`!(`**`v1, v2, r1`**`)`**). Another form of component interaction is through reading and writing of shared data components, which is expressed by the assignment action.

(11) A list of sequential input freeze, port send or subprogram call actions can be expressed with an action sequence (i.e. using character **;** as a separator), whereas a set of simultaneous similar actions can be expressed with an action set (i.e. using character **&** as separator).

(12) The input freeze action specifies that input on ports will be frozen during the execution (see Annex D.5). Freezing of input at dispatch has been discussed in Annex D.4. Once port input is frozen the received value is accessible by naming the port (the core AADL standard refers to it as port variable) and by dequeuing elements from the local port queue (see Annex D.5).

(13) The send actions on ports have been introduced in Annex D.5. Send without value transmits the value previously assigned to a port, either by an assignment action or by a previous send with value. Send with value action assigns the value to the port and then initiates the transmission. Simultaneous send on several ports is expressed as a set of send actions.

(14) Subprogram calls can be specified as action sequences or as action sets. In action sequences they represent synchronous subprogram calls, while in action sets they represent semi-synchronous calls, i.e., multiple calls are initiated to be performed simultaneously. The action set 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 action set.

Note – Synchronous subprograms are bound to the same input/output rules as synchronous action blocks.

(15) Assignment actions assign values to outgoing ports, to out parameters, to local variables, and to persistent state variables, aka, data components in the core AADL model. A data component is referenced either by the data subcomponent name or by the data access feature associated with the data component. When assignment actions are used in action sets, then the assigned values are not accessible to expressions of other assignment actions in the same action set by naming the assignment target.

(16) The type of the assigned value must be consistent with the type of the assignment target. The types are represented by AADL data component classifiers. Basic types, such as integer, float, string, and Boolean, are defined in the Data

Modeling Annex standard AADL package Basic_Types. The corresponding literal values are acceptable values for those types.

(17) Timed actions express consumption of processor cycles (**computation**). Precision of timed actions is implementation defined.

(18) **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.

Syntax

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

behavior_actions ::=
  behavior_action
| behavior_action_sequence
| behavior_action_set

behavior_action_sequence ::=
  behavior_action { ; behavior_action }+

behavior_action_set ::=
  behavior_action { & behavior_action }+

behavior_action ::=
  basic_action
| behavior_action_block
| if ( logical_value_expression ) behavior_actions
  { elsif ( logical_value_expression ) behavior_actions }*
  [ else behavior_actions ]
  end if
| for (
  element_identifier : data_unique_component_classifier_reference
  in element_values ) { behavior_actions }
| forall (
  element_identifier : data_unique_component_classifier_reference
  in element_values ) { behavior_actions }
| while ( logical_value_expression ) { behavior_actions }
| do behavior_actions until ( logical_value_expression )

element_values ::=
  integer_range
| event_data_port_name
| array_data_component_reference

basic_action ::=
  assignment_action
| communication_action
| timed_action
| pause

assignment_action ::=
  target := ( value_expression | any )

target ::=
 | outgoing_port_name
```

```
| outgoing_subprogram_parameter_name
| data_component_reference
| outgoing_port_prototype_name

communication_action ::=
  subprogram_prototype_name ! [ ( subprogram_parameter_list ) ]
| required_subprogram_access_name ! [ ( subprogram_parameter_list ) ]
| subprogram_subcomponent_name ! [ ( subprogram_parameter_list ) ]
| subprogram_unique_component_classifier_reference !
     [ ( subprogram_parameter_list ) ]
| output_port_name ! [ ( value_expression ) ]
| input_port_name >>
| input_port_name ? [ ( target ) ]
| required_data_access_name !<
| required_data_access_name !>
| required_data_access_name . provided_subprogram_access_name !
     [ ( subprogram_parameter_list ) ]
| data_subcomponent_name . provided_subprogram_access_name !
     [ ( subprogram_parameter_list ) ]
 | local_variable_name . provided_subprogram_access_name !
     [ ( subprogram_parameter_list ) ]
| *!<
| *!>

timed_action ::=
  computation ( behavior_time [ .. behavior_time ] )

subprogram_parameter_list ::=
  parameter_label { , parameter_label }*

parameter_label::=
  in_subprogram_parameter | out_subprogram_parameter
in_parameter_value_expression | out_parameter_target

data_component_reference ::=
  data_subcomponent_name { . data_subcomponent_name }*
| data_access_feature_name { . data_subcomponent_name }*
| data_access_feature_name { . data_field }*
| local_variable_identifier { . data_field }*
| data_access_feature_prototype_name { . data_field }*

data_field ::=
   data_subcomponent_name
 | data_access_feature_name
 | data_access_feature_prototype_name

subprogram_parameter ::=
   subprogram_parameter_name {. data_field }*


name ::=
  { thread_group_identifier . }*
  {
    [ subprogram_group_access_identifier . ]
    [ subprogram_group_identifier. ]
    [ feature_group_identifier . ]
  }*
  identifier { array_index }*
name                                                              ::=
  identifier { array_index }*
```

```
array_index ::=
  [ integer_value_variable ]
```

Naming Rules

(N1) The *element_variable_identifier* of a *for* control construct represents a variable whose scope is local to the *for* construct. Such a variable must not be declared as a local variable.

Legality Rules

(L1) In an assignment action, the type of the value expression must match the type of the target.

(L2) An *element_variable_identifier* of a *for* control construct is not a valid *target* for an assignment action.

(L3) The same local variable must not be assigned to in different actions of an action set.

(L4) The same port variable must not be assigned to in different actions of an action set.

(L5) In a subprogram call, the parameter list must match the signature of the subprogram being called.

(L6) When used as a communication action, expression `p?(v)` dequeues the next value from event or event data port `p` and assign it to `v`. If a target is not specified, i.e. `p?`, then dequeued value is lost. If the queue is empty, no action is performed.

(L7) In conformance with the definition of the core AADL Standard, Aa port name only accepts a one dimension array.

(L8) 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 !>).

(L9) In timed actions, the value of the max time must be greater than or equal to the value of the min time.

Semantics

As a result of the above, the sequence of operations of an action block *a* can be represented by an automaton (S,V,T). This automaton extends that of its parent behavior specification by decomposing its sequence of actions into static single assignment (SSA) form and by scheduling all atomic actions (assignments, communications) using execution states.

The semantic function *<<a>>(V,s)= (T,V',s')* associates an action block `a`, starting at execution state *s*, to the corresponding transition function T and returns the state s' that marks the exit point of block `a`. It takes a set of written output ports V as parameter which it returns as V', updated by the set of port names written by block `a`.

- An assignment action *<<x:=e>>(V,s)= ({s-/<x:=e>->s'},V,s')* is a simple transition from s to the fresh name s' of its exit point. It does not update V.
- A pause action is represented by *<<**pause**>>(V,s)=(s-/**pause**!->s',{},s')*
- A port output action *<<p!>>(V,s)=(T,V',s')* is represented by
  - T={s-/p!->s'} and *V'=V ∪ {p},* If `p` does not belong to V.
  - T={s-/p!->s'-/!**pause**->s''} and V'={} if `p` belongs to V.
- A Get_Resource or a Release_Resource action is treated as an implicit communication with the scheduler, and hence in the same manner as an (implicit) output port
- A sequence *<<a;b>>(V,s)=(T U T',V'',s'')* is defined by *<<a>>(V,s)=(T,V',s')* and *<<b>>(V',s')=(T,V'',s'')*

- An conditional *<<**if** g **then** a **else** b>>(V,s)=(T U T' U T''[s'/s''], V' U V'', s')* is defined by *<<a>>(V,t)=(T',V',s')* and *<<b>>(V,t')=(T'',V'',s'')* and *T={s-g->t, s-not g->t'}* where *T''[s'/s'']* stands for the substitution of s'' by s' in T''.
- A while loop *<<**while** g **then** a>>(V,s)=(T U T',V'', s'')* is defined by *<<a>>(V U V',s)= (T,V'',s'')* where *<<a>>(V,s')= (_,V',_)* and T={s-not g->s'', s''-not g->s}
- Until and for loops can be handled in similar manners.

Example

```
-- This example shows the use of non deterministic transitions that enumerate
-- all the possible behaviors of a subprogram, without specifying precisely
-- when each case occurs.

package BA_example4 public
with Base_Types;

subprogram addition
features
  x: in parameter Base_Types::Integer;
  y: in parameter Base_Types::Integer;
  r: out parameter Base_Types::Integer;
  ovf: out parameter Base_Types::Boolean;
end addition;

subprogram implementation addition.default
annex behavior_specification {**
  states
    s0 : initial state;
    s1 : final state;
  transitions
    regular: s0 -[ ]-> s1 { r := x + y ; ovf := false };
    overflow: s0 -[ ]-> s1 { r := 0; ovf := true };
**};
end addition.default;

end BA_example4;
```

Annex D.7 Behavior Expression Language

(1) Behavior expressions consist of logical expressions, relational expressions, and arithmetic expressions. Values of expressions can be variables, constants or the result of another expression.

(2) Behavior expressions have been defined to perform simple calculations only. They do not aim at expressing the full functional logics of the application. In case it is required to express more complex calculations, subprogram call abstractions are recommended.

(3) Variable expression values are evaluated from incoming ports and parameters, local variables, referenced data subcomponents, as well as port count, port fresh, and port dequeue.

(4) Constant expression values are Boolean, numeric or string literals, property constants or property values.

(5) This expression language is derived from ISO/IEC 8652:1995(E), Ada95 Reference Manual §4.4.

(6) Lexical definition of numerals, numeric and string literals is provided by the AADL core standard [AS5506A 15]

(7) Integer values specified by the Behavior Expression Language are equivalent to Base_Types::Integer values as defined in the AADL Data Modeling Annex.

Syntax

```
value ::=
  value_variable
| value_constant
| ( value_expression )

value_variable ::=
  incoming_port_name
| incoming_port_name ?
| incoming_subprogram_parameter
| incoming_port_prototype_name
| incoming_subprogram_parameter_name
| local_variable_name
| data_component_reference
| port_name ' count
| port_name ' fresh

value_constant ::=
  boolean_literal
| numeric_literal
| string_literal
| property_constant
| property_referencevalue

value_expression ::=
  relation { logical_operator relation }*

relation ::=
  simple_expression [ relational_operator simple_expression ]

simple_expression ::=
  [ unary_adding_operator ] term { binary_adding_operator term }*

term ::=
  factor { multiplying_operator factor }*

factor ::=
  value [ binary_numeric_operator value ]
| unary_numeric_operator value
| unary_boolean_operator value

logical_operator ::=
  and | or | xor

relational_operator ::=
  = | != | < | <= | > | >=

binary_adding_operator ::=
  + | -

unary_adding_operator ::=
  + | -
```

```
multiplying_operator ::=
  * | / | mod | rem

binary_numeric_operator ::=
  **

unary_numeric_operator ::=
  abs

unary_boolean_operator ::=
  not

boolean_literal ::=
  true | false

integer_range ::=
  integer_value .. integer_value

integer_value ::=
  integer_value_variable
| integer_value_constant

behavior_time ::=
  integer_value unit_identifier

property_constant ::=
  [ property_set_identifier :: ] property_constant_identifier

property_referencevalue ::=
  # [ property_set_identifier :: ] property_name

property_value_identifier
| component_element_reference # property_name
      { . field_record_property_name }*
 | unique_component_classifier_reference # property_name
      { . field_record_property_name }*

component_element_reference ::=
   subcomponent_identifier
 | local_variable_identifier
 | binded_prototype_identifier
 | feature_identifier
 | self

property_name ::=
   property_identifier { property_field }*

property_field ::=
   [ integer_value ]
 | . item_list_identifier
 | . upper_bound
 | . lower_bound

numeric_literal ::= <refer to [AS5506A 15.4]>

string_literal ::= <refer to [AS5506A 15.5]>

numeral ::= <refer to [AS5506A 15.4.1]>
```

Legality Rules

(L1) Operators have the following precedence, given in increasing order:
- logical operators
- relational operators
- binary adding operators
- unary adding operators
- multiplying operators
- binary numeric operator, unary numeric operator and unary boolean operator

(L2) For a sequence of operators of the same precedence level, the operators are associated with their operands in textual order from left to right. Parentheses can be used to impose specific associations.

(L3) Operand expressions of logical operators must be boolean.

(L4) Operand expressions of relational operators must be of a type that supports comparison.

(L5) Operands of adding, multiplying and other numeric operators must support numeric operations.

(L6) A value expression adopts the type of its top-level operator or single operand.

Examples

```
-- This example gives a basic implementation of a stack data type associated
-- with its access methods. The various corresponding AADL component
-- specifications are encompassed within a same package to offer an Object-
-- Oriented Class style.

package BA_example5 public
with Base_Types;

  data stack
    features
      put : provides subprogram access push;
      get : provides subprogram access pop;
  end stack;

  data implementation stack.default
    subcomponents
      elems : data Base_Types::Integer [100];
      sp : data Base_Types::Integer;
  end stack.default;

  subprogram push
    features
      this : in out parameter stack.default;
      v : in parameter Base_Types::Integer;
      overflow : out event port;
  end push;

  subprogram pop
    features
      this : in out parameter stack.default;
      v : out parameter Base_Types::Integer;
      underflow : out event port;
```

```
  end pop;


  subprogram implementation push.default
  annex behavior_specification {**
    states
      s0 : initial final state;
    transitions
      s0 -[ sp <= 100 ]-> s0 { this.elems[this.sp] := v; this.sp := this.sp+1 };
      s0 -[ sp > 100 ]-> s0 { overflow! };
  **};
  end push.default;

  subprogram implementation pop.default
  annex behavior_specification {**
    states
      s0 : initial final state;
    transitions
      s0 -[ sp > 0 ]-> s0 { this.sp := this.sp-1; v := this.elems[this.sp] };
      s0 -[ sp = 0 ]-> s0 { underflow! };
  **};
  end pop.default;

end BA_example5;

-- Execute conditions are used in the following example which merges sorted
-- data received from two ports. Initially, arrival on either port can
-- trigger a dispatch. Then arrival on one of the ports triggers the next
-- dispatch. Each dispatch consumes one data, which is transferred from one
-- of the two ports. Each transition to one of the two next states completes
-- the execution of the thread, but its memory is supposed to be preserved so
-- that it continues its execution at the next dispatch.

package BA_example6 public
with Base_Types;

thread merger
  features
    p1 : in event data port Base_Types::Integer;
    p2 : in event data port Base_Types::Integer;
    m : out event data port Base_Types::Integer;
end merger;

thread implementation merger.twopersistentstates
subcomponents
  x1 : data Base_Types::Integer;
  x2 : data Base_Types::Integer;
annex behavior_specification {**
  states
    s0 : initial complete state;
    comp : state;
    next1, next2 : complete final state;
  transitions
    s0 -[ on dispatch p1 ]-> next2 { x1 := p1 };
    s0 -[ on dispatch p2 ]-> next1 { x2 := p2 };
    next1 -[ on dispatch p1 ]-> comp { x1 := p1 };
    next2 -[ on dispatch p2 ]-> comp { x2 := p2 };
    comp -[ x1 < x2 ]-> next1 { m!(x1) };
    comp -[ x2 <= x1 ]-> next2 { m!(x2) };
**};
```

```
end merger.twopersistentstates;

-- Next version of the example utilizes the fact that an incoming port holds
-- its previous value and only the port that triggers the dispatch has its
-- input value frozen (updated).

thread implementation merger.portbased
annex behavior_specification {**
  states
    s0 : initial complete state;
    comp : state;
    next1 : complete final state;
    next2 : complete final state;
  transitions
    s0 -[ on dispatch p1 ]-> next2;
    s0 -[ on dispatch p2 ]-> next1;
    next1 -[ on dispatch p1 ]-> comp;
    next2 -[ on dispatch p2 ]-> comp;
    comp -[ p1 < p2 ]-> next1 { m!(p1) };
    comp -[ p2 <= p1 ]-> next2 { m!(p2) };
**};
end merger.portbased;

end BA_example6;
```

Annex D.8 Synchronization Protocols

(1) Thanks to provides subprogram access features, a thread can receive execution requests and execute the corresponding subprogram. With proper statements in a Behavior Annex subclause, it is possible to specify the states where specific requests can be accepted, which correspond to Ada selective accept statements or to HOOD[2] functional activation conditions. This mechanism also allows a clean separation between the functional part of the component defined by a set of subprograms and the synchronization aspects specified by the Behavior Annex automaton. The internal behavior of a server component together with the specification of the interaction protocols between the server component and its clients define the global synchronization aspects.

(2) The `Subprogram_Call_Type` property that is defined in the core standard [AS5506A 5.2] can be used to specify whether results of the requested operation is available to the client when the remote call returns. Possible values are Synchronous (default value) and SemiSynchronous.

(3) The Behavior Annex introduces more precise communication protocols that can be used to better control the blocking duration of a client thread during a remote call to a server thread. These protocols are derived from the main HOOD functional execution requests:

- HSER for Highly Synchronous Execution Request.
- LSER for Loosely Synchronous Execution Request.
- ASER for ASynchronous Execution Request.

(4) These protocols can be specified for each server subprogram access feature of a thread thanks to the `Subprogram_Call_Protocol` property defined by the Behavior_Properties property set:

---

[2] HOOD : Hierarchical Object Oriented Design

```
property set Behavior_Properties is
   Subprogram_Call_Protocol: enumeration (HSER,LSER,ASER) => HSER
   applies to (subprogram access);
end Behavior_Properties;
```

Semantics

(5) The way a server thread responds to a remote subprogram request can be expressed by a transition in the Behavior Annex automata. Such a transition is triggered by a dispatch condition on the corresponding provides subprogram access feature of the server thread. The complete sequence for a server thread to execute a remote subprogram request requires the three following steps:
- step 1: The server automaton is in a state that has an outgoing transition with a dispatch condition on the appropriate provides subprogram access feature.
- step 2: A client thread has sent a call to a requires subprogram access feature properly connected to the provides subprogram access feature of the server thread in order to set the behavior condition of the transition to true.
- step 3: The behavior action of the corresponding transition has been executed.

(6) When a *HSER* `Subprogram_Call_Protocol` is specified, the caller thread remains blocked until the completion of the corresponding behavior action in the server thread, i.e. the sequence described above has reached step 3. Results of the action can be returned to the client (i.e. the `Subprogram_Call_Type` property of the corresponding subprogram classifier is set to *Synchronous*).

(7) When a *LSER* `Subprogram_Call_Protocol` is specified, the caller thread remains blocked until the beginning of the server thread is ready to serve this request, i.e. the sequence described above for a given request has reached step 2. Returning results of the requested action requires a further call from the client. This implies that the value of the `Subprogram_Call_Type` property of the corresponding subprogram classifier is set to *SemiSynchronous*.

(8) When a *ASER* `Subprogram_Call_Protocol` is specified, the caller thread is never blocked by the corresponding remote call. Returning results of the requested action requires a further call from the client. This implies that the value of the `Subprogram_Call_Type` property of the corresponding subprogram classifier is set to *SemiSynchronous*.

(9) When no `Subprogram_Call_Protocol` is specified, default behavior is the one of an *HSER* protocol.

Legality Rules

(L1) When the value of the `Subprogram_Call_Protocol` property is set to *LSER* or *ASER* for a provides subprogram access feature, then the value of the `Subprogram_Call_Type` property of the corresponding subprogram classifier cannot be set to *Synchronous*.

(L2) When the value of the `Subprogram_Call_Protocol` property is set to *LSER* or *ASER* for a provides subprogram access feature, then the corresponding subprogram type cannot have out or in out parameters.

Semantics

The protocol of a call to a subprogram in an action block can be specified using a transition system, by considering the entry and exit state of the call in the client action blocks: *sc,sc'.* All protocols use immediate input-output event port *pc,ps* and intermediate execution state *rc,rs* for the client and server.

- The HSER protocol is encoded by the client transitions sc-/pause,pc->rc and rc-ps->sc' and the server transitions ss-pc->rs and rs-/ps->ss' where the entry and exit state of the server action blocks are *ss,ss'.*
- The LSER protocol is encoded by the client transitions sc-/pause,pc->rc and rc-ps->sc' and the server transitions ss-pc->rs and rs-/ps->ss' where the entry state of the server action blocks is *ss* and ss' is its immediate successor.

- The ASER protocol is encoded by the client transition sc-/rc->pc and the server transition ss-pc->ss' where the entry and exit state of the server action blocks are *ss,ss'*.

If the target subprogram is combinatorial (i.e. it does not dispatch input ports or write output ports) then the client and server action blocks can transit from/to *r* using an immediate transition.

Examples

```
-- Following example shows the use of a LSER call between a client thread
-- and a server thread. The client thread does not need to wait for the
-- completion of the call to a long remote calculation. Result value is later
-- returned thanks to a HSER subprogram call to the server.

package BA_example7 public
with Base_Types;
with Behavior_Properties;

process client_server
end client_server;

process implementation client_server.i
subcomponents
  c : thread a_client;
  s : thread a_server.i;
connections
  access s.long -> c.pre;
  access s.short -> c.post;
end client_server.i;
```

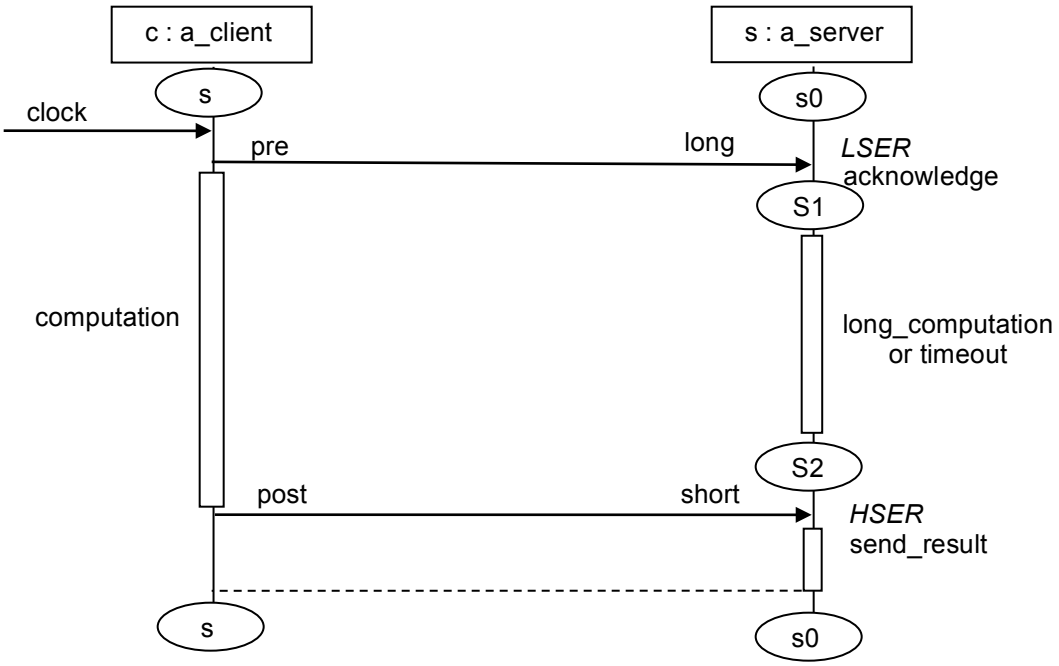

Figure 2: Client Server Behavior

```
thread a_client
features
  pre : requires subprogram access long_computation;
```

```
  post : requires subprogram access send_result;
properties
  Dispatch_Protocol => Periodic;
  Period => 200ms;
annex behavior_specification {**
  variables
    x : result_type;
  states
    s : initial complete final state;
  transitions
    s -[ on dispatch ]-> s { pre!; computation(60ms); post!(x) };
**};
end a_client;

thread a_server
features
  long : provides subprogram access long_computation
    { Behavior_Properties::Subprogram_Call_Protocol => LSER; };
  short : provides subprogram access send_result
    { Behavior_Properties::Subprogram_Call_Protocol => HSER; };
properties
  Dispatch_Protocol => Sporadic;
end a_server;

thread implementation a_server.i
subcomponents
  local_result : data result_type.i;
connections
  access local_result -> long.result;
  access local_result -> short.result;
annex behavior_specification {**
  states
    s0 : initial complete final state;
    s1 : complete state;
    s2 : state;
  transitions
    s0 -[ on dispatch long ]-> s1;
    s1 -[ ]-> s2 { long_computation!; local_result.status := 1 } timeout 60ms;
    s1 -[ timeout ]-> s2 { local_result.status := 0 };
    s2 -[ on dispatch short ]-> s0 { send_result!(local_result) };
**};
end a_server.i;

subprogram long_computation
features
  result : requires data access result_type.i;
  -- LSER subprograms cannot have out parameters
  -- do some long calculation and update result locally
end long_computation;

subprogram send_result
features
  result : requires data access result_type.i;
  output : out parameter result_type;
  -- HSER subprograms can have out parameters
  -- return result to client
end send_result;

data result_type
end result_type;
```

```
data implementation result_type.i
subcomponents
  contents : data Base_Types::Integer;
  status : data Base_Types::Boolean;
end result_type.i;

end BA_example7;
```

Annex D.9 Behavior Abstractions

(1) A transition system can be used to describe the detailed, operational, behavior of any AADL object: thread, process, protocol, scheduler, connection, port, bus, memory, sensor or actuator. Instead, and when a detailed specification is not necessary, or when refinement is necessary, a behavior abstraction can be specified in its stead.

(2) A behavior abstraction is a summary of the input-output behavior of the specified AADL objects. It summarizes the causal and temporal occurrences of events on its input-output ports interface (i.e. its event-driven interface to the environment) and on its timing interface (i.e. its timed triggered interface to a scheduler).

(3) A behavior abstraction is defined by a regular expression on observers and by relations thereof with observers or time.

(4) An observer is a Boolean expression that combines occurrences of events and values of data using logical operators

(5) A regular expression is the sequence, composition, count and iteration of observers

(6) An observer relation associates a pattern of behavior, represented by a regular expression, to a trigger of that behavior, which can either be an observer or time (its period).

Syntax

```
observer ::=
    dispatch_trigger                          -- reference to an input-output port
  | logical_value_expression                  -- logical expression on port values
  | observer ( default | when ) observer      -- disjunction and conjunction

time ::=
  (s | ms | ps )[integer]+integer             -- period and phase

regexp ::=
    observer
  | regexp ; regexp                           -- sequence
  | regexp + regexp                           -- choice
  | regexp | regexp                           -- composition
  | regexp*                                   -- iteration
  | regexp?                                   -- option
  | regexp[n]                                 -- count

behavior_abstraction ::=
  regexp every observer                       -- logical (event-driven) time relation
  regexp every time                           -- real time (time-triggered) relation
```

Semantics

The meaning of a behavior abstraction is the (possibly infinite) set of automata whose traces are recognized by the regular expression of its observers. A behavior annex refines or satisfies a behavior abstraction iff. its automaton is one of these. A controlled behavior annex is one that is specified (or proven) to satisfy a behavior abstraction. The automaton of a controlled behavior annex can be obtained by enforcing the observers of its regular expression using synthesized guarding conditions.
As a result of the above, a controlled behavior is equivalent to an automaton A=(S,X,T,C) where:

- S is its finite set of the initial, execution, complete and final states S°, S^{C}, S^{E} and S^{F}
- X is its finite set of variables V and immediate and delayed input and output ports I and O
- T : S x I -> O x S is the transition system
- C : X -> B is the Boolean function defined on X, which has to equal true at all times.

A transition (s,i,o,s') of T is valid iff. *I(I,V)=true, o(I,V)=(O,V'),* and *C(X)=true.* Just as T is constructed from behavior specifications and action blocks, C is constructed from behavior abstractions.

Computation

TBC .../... HOWTO TYPE/ABSTRACT A BEHAVIOR BY A REGULAR EXPRESSION (OPTIONAL)

Examples

```
process sender                      -- most abstract specification of the sender (p.12)
features
  d: provides data access Integer;
  a: require data access Integer;
annex behavior_specification {**
  a default d
end sender;

process sender                      -- causal specification of the sender (p.12)
features
  d: provides data access Integer;
  a: require data access Integer;
annex behavior_specification {**
    a=0; (a default d=1)
  + a=1; (a default d=0)
end sender;

process sender                      -- timed specification of the sender (p.12)
features
  d: provides data access Integer;
  a: require data access Integer;
annex behavior_specification {**
    ((a=0; (a default d=1) + (a=1; (a default d=0)))
  | ((a every ms[..10[) default (d every ms[10]))
end sender;


process a_client                    -- timed specification of the client-server (p.28)
features
  d: provides data access Integer;
  a: require data access Integer;
annex behavior_specification {**
    prev every ms[200]
```

```
  | post every ms[200]+60
end sender;
```

Annex D.10 Abstract Behavior Relations

Clock Constraint Specifications

An individual event, observer, time scale or a regular expression forms a so-called clock: the logical, regular, repetition of instants. It can be specified and declared using the grammar of `behavior_abstraction` to describe synchronization relations. Moreover, thanks to the properties it inherits from the Kleene algebra it is based on, and that of the automata it abstracts, clock relation can seamlessly be extended to a rich akin to a clock constraint specification language (CCSL, part of OMG's standard MARTE). All clock relations below can be defined from regular expressions on observers and synchronization (**`every`**) except the discrete/continuous relations (with hybrid annex specifications).

Syntax

```
clock ::=
  never                                 -- the null clock
  time                                  -- real-time clock
| regexp                                -- logical clock
| clock when clock                      -- conjunction
| clock default clock                   -- disjunction
| edge port_name                        -- discrete edge of a continuous signal
| port_name sample clock                -- discrete down-sample of a continuous signal
| port_name sustain clock               -- continuous up-sample of a discrete signal

abstract_behavior_relation ::=
  clock every clock                     -- CCSL's synchrony or coincidence
| clock implies clock                   -- CCSL's isubclockof
```

Annex D.11 Refinement-Based Synchronous Behaviors Specification

TBC .../... A COMPLETE CASE-STUDY

Examples

```
process LTTA                -- .../... BASED ON THE LTTA PROTOCOLE
```