

AEROSPACE

STANDARD

AS5506

Issued

Proposed Draft
2006-07

<This page left blank for formatting>

SAE Technical Standards Board Rules provide that: "This report is published by SAE to advance the state of technical and engineering sciences. The use of this report is entirely voluntary, and its applicability and suitability for any particular use, including any patent infringement arising therefrom, is the sole responsibility of the user."

SAE reviews each technical report at least every five years at which time it may be reaffirmed, revised, or canceled. SAE invites your written comments and suggestions.

Copyright © 2004 SAE International

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of SAE.

TO PLACE A DOCUMENT ORDER:

Tel: 877-606-7323 (inside USA and Canada)

Tel: 724-776-4970 (outside USA)

Fax: 724-776-0790

Email: custsvc@sae.org

SAE WEB ADDRESS:

<http://www.sae.org>

Annex BehaviorLanguage Compliance and Application Program Interface

Normative

Annex Behavior.1 Scope

- (1) The purpose of this proposal is to enable the expression of high level composition concepts, such as HRT-HOOD synchronization modes, in AADL, extended by behavioral annexes. As far as possible, we have tried to preserve the semantics and rules of use defined in the AADL standard. The extensions are introduced either via new properties declared in a *property set* or via the annex mechanism.
- (2) The behaviors described in this annex are seen as specifications of the actual behaviors: they can therefore be non-deterministic. They are based on state variables whose evolution is specified by the transitions expressed in the behavioral annex. The data and their types are described in AADL.
- (3) Section annex behavior_specification.2 provides the reference material supporting the concepts and guidelines defined in this annex.
- (4) Section annex behavior_specification.3 defines how data can be specified and its use in the behavioral annex.
- (5) Section annex behavior_specification.4 introduces the syntax of the behavioral annex.
- (6) Section annex behavior_specification.5 defines how a behavior can be attached to a subprogram
- (7) Section annex behavior_specification.6 defines how messages can be sent and received within the annex
- (8) Section annex behavior_specification.7 defines how a passive object and its behavior can be modeled.
- (9) Section annex behavior_specification.8 defines how the access to a shared object can be controlled.
- (10) Section annex behavior_specification.9 defines client/server protocols used by active objects.
- (11) Section annex behavior_specification.10 defines timed behaviors
- (12) Section annex behavior_specification.11 declares an annex specific property set
- (13) Section annex behavior_specification.12 gives the syntax of the behavioral annex.

Annex Behavior.2 References

Annex Behavior.3 Data and Types

Annex Behavior.3.1. Basic types

- (14) Simple types are described in a package containing `data` declarations and include integers, reals and booleans. Compound types such as records can be constructed by the user using `data` hierarchies.

```
package Behavior
public
  data integer
    properties
      Source_Data_Size => 32 bits;
    end integer;

  data float
  end float;

  data boolean
  end boolean;
end Behavior;
```

- (15) The annex can use the usual constants of these types as well as arithmetic and logic operators. We suppose that arithmetic operators are overloaded. More primitive data types could be provided (with different sizes or ranges).

Annex Behavior.3.2. Arrays

- (16) There are no array types in AADL, but we use the notion of multiplicity inherited from UML to declare collections of data which for the present purpose will be considered as ordered.

```
tab: data Behavior::integer { Behavior_Properties::Multiplicity => 10;;}
```

- (17) Arrays dimensions as well as all global parameters can be specified in a property set:

```
property set params is
  tab_size: constant aadlinteger => 10;
end params;
...
tab: data Behavior::integer
  { Behavior_Properties::Multiplicity => value(params::tab_size); };
```

- (18) The use of the multiplicity attribute implies the implicit presence of subprograms giving access to elements of the array. The previous declaration is equivalent to the introduction of a new `data` whose instance would be `tab`:

```
package behavior
...
subprogram integer_10_put
features
  ix: in parameter integer;
  v: in parameter integer;
```

```
this: requires data access integer_10,  
end integer_10_put;  
  
subprogram integer_10_get  
features  
  ix: in parameter integer;  
  v: out parameter integer;  
  this: requires data access integer_10;  
end integer_10_get;  
  
data integer_10  
features  
  put: subprogram integer_10_put;  
  get: subprogram integer_10_get;  
end integer_10;  
end behavior;  
...  
tab: data Behavior::integer_10;
```

(19) The behavioral annex also proposes a syntax to avoid explicit calls to the subprograms giving access to elements in the array.

(20) Multi-dimensional arrays can also be specified. The size of each dimension is defined by the values of the Multiplicities property. For example, the following data declaration defines tab as a two dimension array:

```
tab: data Behavior::integer { Behavior_Properties::Multiplicities =>  
  (2,5); };
```

(21) The two properties are defined in Behavior property set as follows:

```
property set Behavior_Properties is  
  Multiplicity: aadlinteger applies to (data);  
  Multiplicities: list of aadlinteger applies to (data);  
end Behavior_Properties;
```

(22) Elements of an array can be accessed within the annex, either in read mode within expressions or in write mode within assignments:

```
tab[1,2] := tab[1,2] + 1;
```

Annex Behavior.3.3. Enumerated types

(23) The goal of the proposed properties is to support the specification of enumerated data types. An enumerated type is defined as a data classifier. The constants of that type are declared as subprograms having one output parameter of that type. We also consider that enumerated constants can be parameterized.

(24) Enumerated types are specified by a set of injective constructors so that a default implementation can be automatically provided. They extend and generalize the notion of enumerated type by allowing parameters. To each constructor is associated an observer indicating if its parameter has been build using the constructor, and a set of accessors which return the value of the effective arguments passed to the constructor. The following conventions are used to enforce this semantics:

- The boolean property `Behavior_Properties..Abstract` is defined to be true in the data component. Such a data component should not have any AADL implementation. They can be automatically provided by code generators. Its features are restricted to subprograms acting as constructors.
- A constructor has a unique out parameter. Its type must be the data it is associated with.

(25) As an example, we can define the abstract data type *Message* with two constructor, *Success* and *Failure*, the last one taking as parameter an integer which indicates the reason of the failure. In order to ensure modularity, the subprograms and the data components are encapsulated in a package.

```
package message
  subprogram Success
  features
    result: out parameter Message;
  end Success;

  subprogram Failure
  features
    reason: in parameter Behavior::integer;
    result: out parameter Message;
  end Failure;

  data Message
  features
    Success: subprogram Success;
    Failure: subprogram Failure;
  properties
    Behavior_Properties::Abstract => true;
  end Message;
end message;
```

(26) Implicitly declared observers and accessors are specified as follows. By convention, we use the identifier *this* to denote the reference to the data to be accessed.

```
-- observers are named by appending the suffix '?' to the constructor name
subprogram Success?
features
  result: out parameter Behavior_Properties::boolean;
  this: requires data access Message {Required_Access => read_only;};
end Success?;

subprogram Failure?
features
  result: out parameter Behavior::boolean;
  this: requires data access Message {Required_Access => read_only;};
end Failure?;

-- an accessor is defined for each parameter of each constructor

-- this subprogram must only be called on a Failure data. Otherwise, the
result is unspecified
subprogram Failure'Reason
features
  reason: out parameter Behavior::integer;
  this: requires data access Message {Required_Access => read_only;};
```

```
end Failure'Reason,
```

```
data Message
```

```
features
```

```
  Success: subprogram Success;  
  Success?: subprogram Success?;  
  Failure: subprogram Failure;  
  Failure?: subprogram Failure?;  
  Failure'Reason: subprogram Failure'Reason;
```

```
properties
```

```
  Behavior_Properties::Abstract => true;
```

```
end Message;
```

- (27) Automatically generated subprograms can be called in the behavioral annex. As abstract data instances plays the role of values, a functional notation is used for subprogram calls. To each of such subprograms is associated a function with the same name and input parameters. For example, the term `Message.Failure?(Message.Failure(10))` is a legal boolean expression. In the same way, a term can be assigned to a variable as in `err := Message.Failure(10)`, or sent to a data port of the corresponding type.

Annex Behavior.4 Behavior specifications

- (28) This section is an overview of the syntax of the behavioral annex. A more detailed description of the constructs and examples of use are given in the next sections. Behavior specifications can be attached to AADL component implementations using an annex. The behavioral annex describes a transition system (an extended automaton) using six optional sections:

```
annex behavior_specification {**  
  <state_variables>?  
  <initialization>?  
  <states>?  
  <transitions>?  
  <connections>?  
  <composite_declaration>*  
**};
```

- (29) The <state variables> section declares typed identifiers. Types are data classifiers of the AADL model.

```
<state_variables> ::= state variables (<identifier> : <data classifier> ;)+
```

- (30) The state variables must be initialized in the initialization section using a sequence of assignments.

```
<initialization> ::= initial (<assignment> ; )+
```

- (31) The <state> section declares automaton states which can be qualified as **initial**, **complete**, **return**, **urgent** or **composite**.

```
<states> ::= states  
  (<identifier> (, <identifier>)*  
  : (initial | complete | return | urgent | composite)* state ;)+
```

(32) Subprograms and threads start from an initial state. A transition towards a return state ends a subprogram. A transition towards a complete state completes a thread. It will resume from that state at next dispatch. Urgent states are intended mainly for model checking purposes: it is a hint for suggesting that considering interleaving at such states is meaningless. The analysis of such states and the corresponding implementation are considered as tool dependent. Sub-state machines are attached to composite states.

(33) The <transitions> section defines system transitions from a source state to a destination state. The transition can be guarded with events or boolean conditions. An action part can be attached to the transition. It can perform subprogram calls, message sending or assignments. The action is related to the transition and not to the states: if a transition is enabled, the action part is performed and then the current state becomes the transition destination state.

```
<transitions> ::= transitions
  ((<label> :)?
   <state_identifier> -[ <guard> ]-> <state_identifier> { <action>* }; )+
```

(34) The <transitions> section can also define transitions between AADL modes. In this case, the <states> section is forbidden. The enclosing AADL component must not contain a mode automaton has it is replaced by the one defined in the annex.

```
<transitions> ::= mode transitions
  ((<label> :)?
   <mode_identifier> -[ <guard> ]-> <mode_identifier> { <action>* }; )+
```

(35) In a mode transition, the guard and the action parts can only access AADL identifiers that are defined in the mode specified by the source of the transition.

(36) The label contains an optional transition identifier and an optional priority number. Transition priorities allow to fix the evaluation order of transition guards. The evaluation order of two transitions with the same priority is non-deterministic. Transitions with no priority have the lowest priority.

```
<label> ::= <identifier>? ( [ <integer_constant> ] )?
```

(37) The <guard> contains an optional event or event data receipt and an optional boolean condition. The condition may depend on the received data. It has to be noted that, in conformance with AADL execution model, such a data as well as events have already been delivered to the component.

```
<guard> ::=
  [ on <expression> --> ] <event> [ when <expression> ]
  | <expression>
  |
```

(38) The boolean condition can be split in two parts. The **on** part expresses conditions over the current state. The **when** part expresses a condition over the data to be read.

(39) An <event> can be a receipt from an event port (**p?**), from an event data port or a data port (**p? (x)**) where **x** is a state variable or a data subcomponent. It has to be noted that it is also possible to access port directly within expressions. Thus, it is not necessary to first assign them to state variables. An <event> of the form **p?** can also mark the receipt of a server subprogram call.

(40) It is still open to allow or not server subprogram calls inside the guard.

- (41) Basic actions can be assignments, message sending through event, data or event data ports, subprogram calls, and server subprogram calls. It has to be noted that a receipt in an action has not the same semantics as a receipt in a guard: if no messages are present, a receipt in a guard will invalidate the transition, thus allowing other transitions to be chosen while a receipt in an action is erroneous.
- (42) Timing information is mainly provided by the surrounding AADL model, but the annex has a limited support for timing aspects: the **delay** and **computation** actions specify non deterministic waiting time and computation time intervals; the `timeout` boolean constraint can be used inside guards. Timing expressions should be of the `Behavior::time` type. Constants of that type are integers suffixed with AADL Time units.

```

<basic_action> ::=
  computation ( <expression> , <expression> ) ;
| delay ( <expression> , <expression> ) ;
| <communication> ;
| <assignment> ;

<assignment> ::= <expression> := <expression>

<communication> ::=
  <data_port_identifier> (?|!)( <expression> )
| <event_port_identifier> (?|!)
| <subprogram_identifier> ! ( ( <parameter_bindings> ) )?

<parameter_bindings> ::=
  <expression> [ -> <unique_port_identifier> ]
  { , <expression> [ -> <unique_port_identifier> ] } *

```

- (43) Actions may be build from basic actions using a minimal set of control structures allowing sequences, conditionals and finite loops. Finite loops allow iterations over finite integer ranges and over unparameterized enumerated types, which are specified by a data classifier. Sequences of actions are executed in order.

```

<action> ::=
  <basic_action>
| <action> <action>
| if ( <expression> ) <action>
  ( elsif ( <expression> ) <action> )* ( else <action> )? end if ;
| for ( <identifier> in <integer_range> ) { <action> } ;
| for ( <identifier> in <data_classifier> ) { <action> } ;

```

- (44) Legal expressions are arithmetic and boolean compositions of basic expressions that include:

- feature names (data and parameters) of the current component type,
 <basic_expression> ::= <feature_identifier>
- access to `fresh` and `count` attributes of an input port of the current component type,
 <basic_expression> ::= <feature_identifier>'[fresh | count]
- data access features and data subcomponents
 <basic_expression> ::=
 <data_access_identifier>

| <data_subcomponent_identifier>

- subcomponents of data access features and data implementation subcomponents

<basic_expression> ::=

<data_access_identifier>.<data_subcomponent_identifier>

| <data_subcomponent_identifier>.<data_subcomponent_identifier>

- property constants, accessed through the value property operator,

<basic_expression> ::= **value** (<property_constant_identifier>)

- integer literals with an optional time unit suffix denoted time expressions

<basic_expression> ::= <integer_literal> <time_unit> ?

- state and loop variables

<basic_expression> ::=

<state_variable_identifier> | <loop_variable_identifier>

- array access,

<basic_expression> ::= <array_identifier> [<expression>]

- terms specified by abstract data types.

<basic_expression> ::=

<data_classifier_unique_identifier>.<subprogram_feature>
(<expression> (, <expression>) *)

| <data_classifier_unique_identifier>.<subprogram_feature>

| <data_classifier_unique_identifier>.<subprogram_feature>? (<expression>)

| <data_classifier_unique_identifier>.<subprogram_feature>?

| <data_classifier_unique_identifier>.<subprogram_feature>'

<parameter_identifier> (<expression>)

(45) A priority rule must be defined to specify the visibility of identifiers. It is the following: loop variables, state variables, data subcomponents, and features.

(46) The `connections` section extends the one which is already present in AADL and allows the specification of links between entry points and their corresponding implementations. It also extends parameter passing declarations. More precisely, it allows the following declarations

- link between a server subprogram and a call occurrence: it specifies which subprogram is called when the call is served. The call occurrence allows the specification (via AADL connections) of call parameters which can be provided partly by the client and partly by the server thread.
- link between an input event port and a call occurrence: it specifies which subprogram is called when the event is handled. An event data port can provide a parameter to the subprogram. Other parameters or data accesses can be provided by the server.
- link between a parameter of a server subprogram and a parameter of a call occurrence: it specifies how parameters provided by the client are transmitted from the server entry point to the called subprogram. The connection between the client and the server subprogram is not considered here and will be specified by a connection in the next version of AADL.

The use of these connections is often useless if naming conventions are applied.

(47) A sub state-machine can be attached to a composite state or to an AADL mode. States can be decomposed hierarchically as in statecharts.

<composite_declaration> ::=

composite history ? [**state** <state_identifier> | **mode** <mode_identifier>]

```

<states>
<transition>
<composite_declaration>*
end [ <state_identifier> | <mode_identifier> ] ;

```

- (48) If a composite state is declared with the **history** attribute, the next transition to this state will enter the last visited substate. A first visit to the composite state will enter one of its initial substates
- (49) Substates can be declared with the return or complete attributes. Entering such states terminates the specified behavior.
- (50) A composite state cannot be marked as complete or return.
- (51) Transitions applying to a composite state also apply to each of its substates.
- (52) Possibly composite substates and their corresponding transitions can be attached to an AADL mode. This automaton describes the mode dependent behavior of the enclosing AADL component.

Annex Behavior.5 Subprogram behavior

- (53) A behavior expressed by the annex can be attached to a subprogram implementation. The behavior can refer to the subprogram parameters (according to their mode) as well as to local variables and to the visible global variables declared in AADL.

Annex Behavior.5.1. Specification of a subprogram behavior

- (54) The automaton specifying the subprogram implementation has one or more **return** states indicating the return to the caller. The parameters in IN and OUT mode can be read and modified respectively (values are transmitted before the call for the parameters in IN mode and during the transition to a *return* state for the OUT mode).
- (55) When it describes the specification of an actual implementation, the automaton can be non-deterministic.
- (56) Transitions can be named. Transition names (normal and overflow in the following example) will be used in the behavioral property language.

```

subprogram addition
features
  x: in parameter Behavior::integer;
  y: in parameter Behavior::integer;
  r: out parameter Behavior::integer;
  ovf: out parameter Behavior::boolean;
end addition;

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

```

- (57) The body of a subprogram can also send an event.

```

subprogram addition

```

features

```
x: in parameter Behavior::integer;
y: in parameter Behavior::integer;
r: out parameter Behavior::integer;
ovf: out event port;
end addition;

subprogram implementation addition.default
annex behavior_specification {**
states
  s0 : initial state;
  s1 : return state;
transitions
  s0 -[ ]-> s1 { r := x + y; };
  s0 -[ ]-> s1 { ovf!; };
**};
end addition.default;
```

Annex Behavior.5.2. Invocation of subprograms

(58) While the AADL control flows defines the call sequences produced by a subprogram, the annex enables us to express dependencies between the control flows and state variables or parameters. A subprogram specification can express other subprogram calls or notification of events.

(59) The example below shows that the `print` function is called up when the `debug` parameter is true.

```
subprogram start_read
features
  debug: in parameter Behavior::boolean;
end start_read;

subprogram implementation start_read.i

calls {
  p1: subprogram std::print;
};

annex behavior_specification {**
states
  s0 : initial state;
  s1 : return state;
transitions
  s0 -[ on debug ]-> s1 { std::print!; };
  s0 -[ on not debug ]-> s1 {};
**};
end start_read;
```

(60) Parameters can be passed to called subprograms. Formal parameters binding is defined using the notation:

```
actual -> formal.

annex behavior_specification {**
state variables
  result: Behavior::integer;
```

```

states
  s0 : initial state;
  s1 : return state;
transitions
  s0 -[ ]-> s1 { addition!(1->x,2->y,r->result); };
  **};

```

- (61) The input / output parameters of the invoked subprogram can also be transmitted between call occurrences. A fixed data flow graph can be specified in AADL:

```

subprogram test
features
  x: in parameter Behavior::integer;
  y: in parameter Behavior::integer;
  z: out parameter Behavior::integer;
end test;

subprogram implementation test.default
- AADL standard: specification of call occurrences
calls {
  add1: subprogram addition;
  add2: subprogram addition;
};
- fixed data flow graph
connections
  parameter x -> add1.x;
  parameter y -> add1.y;
  parameter add1.r -> add2.x;
  parameter y -> add2.y;
  parameter add2.r -> z;
end test.default;

```

- (62) Whereas it is only possible to declare mode dependent linear flow graphs in AADL, the annex can be used to specify data dependent flow graphs. State variables can also be declared in the annex and used as local variables.

```

subprogram implementation test.default
annex behavior_specification {**
  state variables
    aux : Behavior::integer;
  states
    s0: initial state;
    s1: state;
    s2: return state;
  transitions
    s0 -[ ]-> s1 { addition!(x->x,y->y,r->aux); };
    s1 -[ on aux < 10 -] -> s2 { addition!(aux->x,y->y,r->z); };
    s1 -[ on aux >= 10] -> s2 { z := aux; };
  **};
end test.default;

```

Annex Behavior.6 Sending / receiving messages

(63) Messages are received through event, data or event data ports. Event and event data ports are associated to queues. On dispatch, zero, one or all elements of the queue are transferred to the thread, depending on the value of the *Dequeue_Protocol* property. If it has the *AllItems* value, then all the queued messages are stored in an internal queue and can be accessed by the behavioral annex using the dequeue operator on the port name. The old contents of the queue is lost. Otherwise, it has the value *OneItem* and one message is popped from the queue and transferred to the thread. For data ports, zero or one message is transferred. If no new messages are transferred, the old contents is seen by the annex and the *fresh* attribute associated to the port is set to *false*. Single data is read using the port name. Each access to internally queued event/data dequeues it from the internal queue. The following constructs are thus available on an input port *p*:

- *p* can be used as a data value and returns the data stored in the port variable if *p* is a data port or an event data port with the *OneItem* Dequeue_Protocol. The value can be written if the port direction is `out` or `in out`.
- *p'count* returns the number of internally queued messages
- *p'fresh* return true if the port variable has been refreshed at the previous dispatch.
- *p?* dequeues an event on an event port variable. *p'count* must be non zero and is decremented.
- *p?x* dequeues in the variable *x* a data on an event data port variable with the *AllItems* protocol. *p'count* must be non zero and is decremented.

(64) According to the AADL execution model, the number of queued event/data transferred depends on the presence of a connection on a special *Dispatch* port. If the dispatch port is connected, data present on all input ports are transferred, otherwise, only data present on the dispatching port is transferred and the corresponding entry point is called. In order make these behaviors more explicit, we introduce a new boolean property named *With_Dispatch*, which, when set, specifies that the Dispatch port should be connected.

```
property set Behavior_Properties is
  With_Dispatch: aadlboolean applies to
  (thread);
end Behavior;
```

(65) Messages are sent through output event, data or event data ports. A data can be stored in data and event data ports using the port name as a variable. It is implicitly transferred after completion. An event can be sent immediately to an event or event data port using the *p!* notation. The *Raise_Event* system call is called and the data is transferred if *p* is an event data port. On event data ports, both notations can be used so that *p!d* writes *d* to the port and sends an event. To sum up, the following statements are defined:

- *p!* calls *Raise_Event* on an event or event data port. The event is immediately sent to the destination with the stored data if any.
- *p := d* writes data *d* on a data or event data port. Data is transferred to the destination port at the next *p!* call for event data ports or after completion for data ports.
- *p!d* writes data *d* to the event data port *p* and calls *Raise_Event*. The data is immediately sent to the destination.

- (66) If the input or the output is a port group, the parameters are data received from or sent to each port in the group. The use of parentheses separates the sub-groups from the hierarchical groups. For homogeneity purposes, the void value () can be received from / sent to an event port.
- (67) Message receipt as well as calls to server subprograms¹ can be performed in the guard part or in the action part. A non deterministic choice is performed between transitions starting from the current state and that are ready to be fired. However, the actual implementation, which must refine the behavioral specification can be deterministic.
- (68) Message sending on output ports is always non blocking and is performed in the action part. If present, the data value is written into the output port. A *RaiseEvent* call is implicitly performed if a message is sent to an event or event data port. An event must be sent to event data ports in order to transmit their data value immediately. Otherwise, the data value is transmitted at completion or deadline.
- (69) If the protocol of a server subprogram is not ASER, the call may block. Thus, alternative transitions with ready guards will be selected in priority to a transition containing a blocking call in its guard.
- (70) The following example illustrates reception/emission on event data ports.

```

thread test
features
  p_in: in event data port Behavior::integer;
  p_out: out event data port Behavior::integer;
end test;

thread implementation test.default
subcomponents
  x: data Behavior::integer;
annex behavior_specification {**
  states
    s0: initial complete state;
  transitions
    s0 -[p_in?(x)]-> s0 { p_out!(x+1); };
  **};
end test.default;

```

Annex Behavior.7 Passive objects

- (71) Passive objects are translated directly to AADL data components. A data declaration corresponds to a class declaration encapsulating data and exporting access methods. The profile of methods must be declared outside the component via a subprogram component type. The actual object is provided via a data access feature named **this**. The data component and the subprograms defining the method profiles can be grouped together in a package.

```

package stack
public
  subprogram push
  features
    v: in parameter Behavior::integer;
    this:requires data access stack {Required_Access => access Read_Write;};
    overflow: out event port;
  end push;

```

¹ The semantics of server subprogram calls within guards needs not be clarified.

```

subprogram pop
  features
    v: out parameter Behavior::integer;
    this:requires data access stack {Required_Access => access Read_Write;};
    underflow: out event port;
end pop;

subprogram clear
  features
    this:requires data access stack {Required_Access => access Read_Write;};
end clear;

subprogram is_empty
  features
    v: out parameter Behavior::boolean;
    this: requires data access stack {Required_Access => access Read_Only;};
end is_empty;

data stack
  features
    put: subprogram push;
    get: subprogram pop;
    clear: subprogram clear;
    is_empty: subprogram is_empty;
end stack;
end stack;

```

(72) The internal state of a data component is specified by the subcomponents of the data implementation. It is accessed through the **this** data reference by the behavioral annex associated to the implementation of each subprogram. Dereferencing **this** is implicit.

(73) When arrays are declared using the *Multiplicity* property, array elements can be accessed using the usual bracket notation. Array elements are numbered from 1.

```

package stack
private
  data implementation stack.default
    subcomponents
      elems:data Behavior::integer
        { Behavior_Properties::Multiplicity => value(params::tab_size);};
      sp: data Behavior::integer;
    end stack.default;

  subprogram implementation push.default
  annex behavior_specification {**
  states
    s0 : initial return state;
  transitions
    s0 -[ on sp <=value(params::tab_size)]-> s0 { elems[sp]:=v; sp:=sp+1; };
    s0 -[ on sp > value(params::tab_size)]-> s0 { overflow!; };
  **};
  end push.default;

  subprogram implementation pop.default
  annex behavior_specification {**

```

```

states
  s0 : initial return state;
transitions
  s0 -[ on sp > 0 ]-> s0 { sp := sp - 1; v := elems[sp]; };
  s0 -[ on sp = 0 ]-> s0 { underflow!; };
  **};
end pop.default;
...
end stack;

```

- (74) Contrary to abstract data types defined by a set of constructors and acting as values, objects have an internal state that can be accessed by several threads. Subprograms can modify the referenced object. Thus, access to such data instances must be controlled.

Annex Behavior.8 Protected objects

- (75) AADL supports the notion of protected object: a protected object is a data component whose `Concurrency_Control_Protocol` property is defined. The access control mechanisms are left open by AADL and must be defined via the `Supported_Concurrency_Control_Protocols` property. Allowed values will depend on the analysis and code generation tools.

- (76) Taking the example of the stack, we will add a property to the specification of the data component:

```

data stack
features
  put: subprogram put;
  get: subprogram get;
  empty: subprogram empty;
  full: subprogram full;
properties
  Concurrency_Control_Protocol => Maximum_Priority; -- for example
end stack;

```

Annex Behavior.9 Thread dispatch

- (77) Threads interact through shared data and ports. 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 thread is dispatched. If a specific port called the *Dispatch* port is connected and receives an event, one or all of the queued event/data of all ports are transferred. Otherwise, one or all of the event/data of one of the ports that have received an event is transferred. Then, the thread is dispatched and executes the behavior associated to the Dispatch port or to the selected port. We propose to extend this mechanism in order to be able to specify a subset of enabled ports which can trigger a dispatch and of which contents is transferred to the thread. Two modes are thus proposed:

- (78) if the Dispatch port is connected and receives an event, event/data available on the subset of the ports are transferred and the entry point associated to the Dispatch port is called to process transferred data.
- (79) If the Dispatch port is not connected and an event arrives to a port of the subset, queued event/data of that port is transferred and the corresponding entry point is called. The choice is non deterministic among the most urgent ports of the subset that have received an event.

- (80) The subset of enabled ports is defined at initialization and at completion and controls the next dispatch. The subset can be specified using a new predeclared runtime service

```
Enable_ports: subprogram;
```

The call to this runtime service is implicit when using the proposed annex.

Annex Behavior.10 Active objects

- (81) Active objects have their own behavior defined by threads. They also encapsulate data and accept requests from their environment. The thread can receive requests and call the corresponding subprogram. It thus specifies states where specific requests can be accepted, which corresponds to guarded acceptance of Ada or to HRT-HOOD functional activation conditions. This mechanism also allows a clean separation between the functional part of the object defined via a set of subprograms (the object methods) and the synchronization aspects. The behavior of the object together with the specification of the interaction protocol between the object and its environment define the synchronization aspects.

- (82) As it is proposed by HRT-HOOD, three atemporal protocols are considered: asynchronous (ASER), synchronous (HSER) and semi-synchronous (LSER). They are introduced by the following property declaration:

```
Server_Call_Protocol: type enumeration (ASER, HSER, LSER) => HSER  
    applies to (server subprogram);
```

- (83) We can characterize the three protocols as follows:

- With the ASER protocol, there is no synchronization
- With the LSER protocol, the caller waits for the acceptance of the request.
- With the HSER protocol, the caller waits for the completion of the request and gets results if any.

- (84) In the following, an active object exports a set of so called entry points. These can be seen as an extension of AADL server subprograms. In fact, a protocol is associated to an entry through a property. The parameters and the results of subprograms are handled automatically: the actual call to the subprogram is implicit.

- (85) In the behavioral annex, the transitions of the server thread specify when requests are accepted and when the client is resumed.

- (86) The subset of enabled event or event data ports that are concerned by data transfer and that can trigger a dispatch if the *Dispatch* port is not connected is specified as follows: a port is in the enabled subset if a transition starting from the current state and the guard of which evaluates to true contains a *reception* from that port.

- (87) At initialization, the current state is an initial state. When, reaching a complete state, the thread is completed. The next dispatch starts from this completion state, which is used to determine the enabled ports. The thread is supposed to retain its data between a completion and the following dispatch. Reaching a complete state can be interpreted as calling the *Await_Dispatch* service.

- (88) In the same way, server subprograms can be guarded. The guard should only depend on the internal state of the thread. It cannot depend on the parameters of the subprogram.

Annex Behavior.10.1. Asynchronous client-server protocol

- (89) In the asynchronous protocol several clients can invoke the services of a server object. The invocation is non-blocking. The processing of an accepted request is carried out in parallel with the client. The entry points are represented by event ports which may support data. These events are stored in queues that can be bound via the `Queue_Size` attribute.
- (90) Subprogram calls can be attached to event or event data input ports of non periodic threads. The subprogram attached to a port is automatically executed by the thread if an event is received on that port. It is specified by an extension of the `connection` clause defined in the annex.
- (91) We use AADL call sequences and associated connections to link the port with its corresponding subprogram features. The subprogram associated to an event data port takes one input parameter of the same type as the port. The subprogram associated to an event port takes no parameters. Output ports of the thread must be declared within the subprogram. They are linked together by the AADL connection clause.

```

thread test
features
  p_in: in event data port Behavior::integer;
  p_out: out event data port Behavior::integer;
end test;

subprogram get
features
  v : in parameter Behavior::integer;
  p_out: out event data port Behavior::integer;
end get;

thread implementation test.default
calls {
  g: subprogram get;
};
connections
  parameter p_in -> g.v;
  event data port g.p_out -> p_out;
annex behavior_specification {**
connections
  event data port p_in -> g; -- g handles events received from p_in
**};
end test.default;

subprogram implementation get.default
annex behavior_specification {**
states
  s0: initial return state;
transitions
  s0 -[]-> s0 { p_out!(v+1); };
**};
end get.default;

```

(92) Subprograms associated to input ports can access shared data declared within the thread. The following example describes a server monitoring accesses to a resource in order to check its use. It uses a counter of actual read accesses and a flag indicating if there is a write access. An error is signaled if the required access is illegal (a write with ongoing read or writes, a read with ongoing writes). First, we give such a description using AADL event ports, then we introduce server subprograms to which we associate the ASER protocol.

```

subprogram start_read
  features
    rd: requires data access Behavior::integer;
    wr: requires data access Behavior::boolean;
    err: out event port;
  end start_read;

subprogram implementation start_read.default
  annex behavior_specification {**
  states
    s0: initial return state;
  transitions
    s0 -[on not wr]-> s0 { rd := rd + 1; }; -- legal read
    s0 -[on wr]-> s0 { err!; }; -- illegal read detected
  **}
  end start_read.default;
  ...
  thread RW_monitor
  features
    start_read: in event port;
    end_read: in event port;
    start_write:in event port;
    end_write: in event port;
    access_error: out event port;
  end RW_monitor;

  thread implementation RW_monitor.i
  subcomponents
    reading_cpt: data Behavior::integer;
    writing_flg: data Behavior::boolean;
  calls
    { sr: subprogram start_read; };
    { er: subprogram end_read; };
  connections
    data access reading_cpt -> sr.rd;
    data access writing_flg -> sr.wr;
    event data port sr.err -> access_error;
    ...
  annex behavior_specification {**
    connections -- from server input ports to call occurrences
    event port start_read -> sr;
    event port end_read -> er;
  **};
  end RW_monitor.i;

```

(93) For homogeneity purposes, we propose the following alternative solution where event ports are replaced by server subprograms associated with the ASER protocol specifying an asynchronous call.

By convention, we suppose that the subprogram called when a request is received is called from a point with the same name as the server entry.

```

thread RW_monitor
  features
    start_read  : server subprogram start_read
                  { Behavior_Properties::Server_Call_Protocol => ASER; };
    end_read    : server subprogram end_read
                  { Behavior_Properties::Server_Call_Protocol => ASER; };
    start_write : server subprogram start_write
                  { Behavior_Properties::Server_Call_Protocol => ASER; };
    end_write   : server subprogram end_write
                  { Behavior_Properties::Server_Call_Protocol => ASER; };
    access_error: out event port;
  end RW_monitor;

thread implementation RW_monitor.i
  subcomponents
    reading_cpt: data Behavior::integer;
    writing_flg: data Behavior::boolean;
  calls-- subprograms that are called on server requests
    { start_read: subprogram start_read; };
    { end_read: subprogram end_read; };
  connections
    data access reading_cpt -> sr.rd;
    data access writing_flg -> sr.wr;
    event data port sr.err -> access_error;
    ...
annex behavior_specification {**
  connections -- from server entry points to call occurrences
    subprogram start_read -> start_read; -- implicit
    subprogram end_read -> end_read; -- implicit
**};
end RW_monitor.i;

```

(94) The previous encoding does not support the expression of activation conditions. For this purpose, we add a behavior to the thread itself, which allows the selection of entries. The thread specifies which entries can trigger a dispatch. The thread is dispatched when data arrives on one of the enabled input event/data ports and the minimal inter arrival time has elapsed.

(95) Activation conditions are used in the following example which merges sorted data received from two ports. Initially, arrival on both ports 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.

```

thread merger
  features
    p1 : in event data port Behavior::integer;
    p2 : in event data port Behavior::integer;
    m  : out event data port Behavior::integer;
  end merger;

thread implementation merger.i
  annex behavior_specification {**

```

```

state variables
  x1 : Behavior::integer;
  x2 : Behavior::integer;
states
  s0 : initial state;
  comp : state;
  next1, next2 : complete state;
transitions
  s0 -[ p1?(x1) ]-> next2 { };
  s0 -[ p2?(x2) ]-> next1 { };
  next1 -[ p1?(x1) ]-> comp { }; -- only one enabled port
  next2 -[ p2?(x2) ]-> comp { }; -- only one enabled port
  comp -[ on x1 < x2 ]-> next1 { m!(x1); };
  comp -[ on x2 <= x1 ]-> next2 { m!(x2); };
  **);
end merger.i;

```

(96) Subprograms can also be attached to input ports. The thread behavior specifies acceptance conditions for subprograms. Thus, we separate the action performed on data and the selection of the entry. If *sp* is a subprogram declared as a feature of the thread, *sp?* specifies that the thread waits for a dispatch on that port. On dispatch, the corresponding subprogram is called and the action part of the transition is executed.

(97) The next *shuffle* example uses the server subprogram features with the ASER protocol to emit data received alternatively from one of its two input ports.

```

subprogram put
  features
    x: in parameter Behavior::integer;
    m: out event data port Behavior::integer;
end put;

subprogram implementation put.i
  annex behavior_specification {**
    states
      s0 : initial return state;
    transitions
      s0 -[ ]-> s0 { m!(x); };
    **);
end put.i;

thread shuffle
  features
    put1 : server subprogram put.i
           {Behavior_Properties::Server_Call_Protocol => ASER;};
    put2 : server subprogram put.i
           {Behavior_Properties::Server_Call_Protocol => ASER;};
    m : out event data port Behavior::integer;
end shuffle;

thread implementation shuffle.i
  calls
    { put1: subprogram put.i; };
    { put2: subprogram put.i; };
  connections

```

```

event data port put1.m -> m;
event data port put2.m -> m;
annex behavior_specification {**
  states
    s1 : initial complete state;
    s2 : complete state;
  transitions
    s1 -[ put1? ]-> s2;
    s2 -[ put2? ]-> s1;
  connections -- from server parameters to call occurrence parameters
    parameter put1.m -> put1.m; -- implicit
    parameter put2.m -> put2.m; -- implicit
**};
end shuffle.i;

```

Annex Behavior.10.2. Highly synchronous client-server protocol

- (98) The highly synchronous client-server protocol enables the service which has been called up to return a result to the client. It corresponds to a remote procedure call and to the HSER mode of HRT-HOOD. In AADL, this behavior corresponds to the client-server mode which however does not support the specification of activation conditions.
- (99) The operation activation constraints can be expressed via a 3-way synchronization (the client, the server thread and the called service). These conditions are expressed by the server's behavior automaton. As for event (data) ports, a receipt of a remote procedure call can trigger a dispatch only if the entry is enabled, which is specified by the guard of an acceptance transition, i.e. a transition of which guard contains the name of the entry point. If the entry is selected for dispatch, a call to the corresponding subprogram is implicitly performed. When it returns, the client gets the result and resumes its execution while the server executes the action part of the transition. The server thread continues its execution until it reaches a complete state. At this point, next enabled entries are determined for the next dispatch.
- (100) The following example defines a server with two entry points using an internal buffer. It is possible to remotely call the `put` operation (resp. `get`) if the buffer is empty (resp. full). When an entry point is called and the corresponding condition is satisfied, the associated data subprogram is called. This example illustrates a new feature: the client interface differs from the server interface : the implementation of the subprograms need to access a shared data, the actual buffer, which is not seen by the caller.

```

subprogram put
  features
    v: in parameter Behavior::integer;
    this: requires data access buffer;
  end put;

subprogram client_put
  features
    v: in parameter Behavior::integer;
  end client_put;
...
data buffer
  features
    put: subprogram put;
    get: subprogram get;
  end buffer;

```

```

thread BufferServer
  features
    put: server subprogram client_put;
    get: server subprogram client_get;
  end BufferServer;

thread implementation BufferServer.default
  subcomponents
    buf : data buffer;
  calls { put: subprogram buffer.put; };
        { get: subprogram buffer.get; };
  connections -- should be considered as implicit when using 'this'
    data access buf -> put.this;
    data access buf -> get.this;
  annex behavior_specification {**
  states
    empty: initial complete state;
    full : state;
  transitions
    empty -[ put? ]-> full;
    full  -[ get? ]-> empty;
  connections
    subprogram put -> put; -- implicit
    subprogram get -> get; -- implicit
    parameter put.v -> put.v; -- implicit (identical parameter names)
    parameter get.v -> get.v; -- implicit (identical parameter names)
  **};
end BufferServer.default;

```

Annex Behavior.10.3. Semi-synchronous protocol

- (101) In the semi-synchronous protocol, the client is unblocked when the message is accepted. In this mode, only the input parameters are transmitted. The thread is dispatched when it receives a call on an enabled entry. At that moment, the client is resumed and the server thread implicitly calls the entry point.
- (102) The semi-synchronous and synchronous protocols are distinguished by the client wakeup time (which occurs when the request is accepted for the semi-synchronous mode and at the end of processing for the synchronous mode). This protocol is specified by attaching the **LSER** property to the subprogram declaration.
- (103) The following example declares the semi-synchronous entry point `put`. The client is woken up before the effective subprogram call by the server thread.

```

thread BufferServer
  features
    put: server subprogram buffer.put
        {Behavior_Properties::Server_Call_Protocol => LSER;};
  end BufferServer;

thread implementation BufferServer.default
  annex behavior_specification {**
  states

```

```
s0 . initial state,
transitions
  s0 -[ put? ]-> s0;
**}
end BufferServer.default;
```

Annex Behavior.11 Link with AADL execution model

- (104) The special AADL ports Dispatch, Complete and Error can be used within the annex.
- (105) Events and data read from input ports contain information transmitted to the thread at dispatch time. Messages received on a port after dispatch cannot be seen. However, if multiple data are transferred on dispatch (using the *AllItems* property value), they are stored in internal queues and can be processed.
- (106) It is possible to get the number of (data) events that can be read from the internal queue of a port using the expression

port_identifier' **count**.

The count attribute is initialized when the thread is dispatched. Each execution of the dequeue operator *p?* decrements the count attribute of the port.

- (107) These features are illustrated through an example taken from AADL standard. It exploits the AADL execution model to compute a speed by counting the number of ticks received each second by the measuring thread. At every period, events received during the previous period are transferred to the thread and become available to the user.

```
thread speed
  features
    tick: in event port { Dequeue_Protocol => AllItems; };
    sp: out data port Behavior::integer;
  properties
    Dispatch_Protocol => periodic;
    Period => 1 s;
end speed;

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

- (108) As data ports can contain data which have not been updated at the previous dispatch, it is possible to check if data is fresh using the boolean expression:

port_identifier' **fresh**

- (109) Data written to output data ports are transmitted to the destination ports after completion of the thread. Only the last written value is transferred.

- (110) An implicit *RaiseEvent* call is immediately performed when outputs are done on event (data) ports through the ! Operator. It has to be noted that two consecutive send to the same port can be interleaved with messages from other threads. If two writes are performed to an event data port without interleaving a *RaiseEvent* call (through the ! operator), the first write is useless.
- (111) Calls to methods of shared data are implicitly encapsulated by calls to *GetResource* and *ReleaseResource*.
- (112) Enabled ports for the next dispatch are automatically computed at completion. They are input event or event data ports, or server subprograms.

Annex Behavior.12 Timing Aspects

- (113) All timing constants are defined by expressions of the `Behavior::Time` type. Constants of that type can be either references to AADL property constants or integer literals suffixed by a time unit. Arithmetic operations are allowed on time values and follow usual laws on units.

Annex Behavior.12.1. Elapse of time

- (114) Apart from assignments, it is possible to associate timed actions with a transition:
- (115) `Computation(min,max)` expresses the use of the CPU for a non-deterministic period of time between `min` and `max`.
- (116) `Delay(min,max)` expresses a suspension for a non-deterministic period of time between `min` and `max`.

Annex Behavior.12.2. Periodic Threads

- (117) A periodic thread is declared using pre-declared AADL attributes:
- `Dispatch_Protocol => periodic`, and
 - `Period`
- (118) The behavior described by the annex is triggered from its *initial* state and must reach a *complete* state before the `Compute_Deadline`. Subsequent dispatches will resume the thread from the last *complete* state.

```
thread Transmitter
features
  lput: subprogram put;
end Transmitter;

thread implementation Transmitter.default
properties
  Dispatch_Protocol => periodic;
  Period => 10 ms;

  annex behavior_specification {**
  states
    s0: initial complete state;
  transitions
```

```
s0 -[ lput! ]-> s0 {},
**);
end Transmitter.default;
```

Annex Behavior.12.3. Interaction in bounded time

(119) Several techniques can be used to bound the waiting time for a synchronization when a subprogram is called, depending on the declaration to which the attribute specifying a time boundary is attached:

- Subprogram declaration, as it is proposed by HRT-HOOD: the waiting time of each call to the entry point is bounded by the same constant
- Subprogram call: a maximal amount of time waiting for is specified for each call occurrence

(120) The solution we have chosen to express bounded wait involves adding a *timeout* T predicate which becomes true when the system has been waiting for T units of time since the last transition has been completed. Using such a guard on an alternative transition limits the time the response of a server subprogram can be called. Given that the transitions are fired as soon as possible, a synchronization with timeout is expressed as follows:

```
s0 -[ on g --> p! ]-> s1 { };
s0 -[ on g and timeout T ]-> s2 { };
```

The synchronization on *p!* is then expected for *T* units of time at the most. This declaration is meaningful if the communication is time consuming. This is the case for calls to server subprograms using the HSER or LSER protocols.

Annex Behavior.13 Predefined types and properties

```
package Behavior
public
  data integer
  properties
    Source_Data_Size => 32 bits;
  end integer;

  data float
  end float;

  data boolean
  end boolean;
end Behavior;

property set Behavior_Properties is
  Server_Call_Protocols: type enumeration (HSER, LSER, ASER);
  Server_Call_Protocol : Behavior_Properties::Server_Call_Protocols => HSER
  applies to (server subprogram);
  Multiplicity : aadlinteger applies to (data);
  Multiplicities : list of aadlinteger applies to (data);
  Abstract : aadlboolean applies to (data);
  With_Dispatch: aadlboolean applies to (thread);
end Behavior_Properties;
```

Annex Behavior.14 Syntax of the annex

```

behavior_annex ::=
  [ state variables {behavior_variable_decl}* ]
  [ states {behavior_state}* ]
  [ initially (assignment ;)+ ]
  [ mode ? transitions {behavior_state_transition}* ]
  [ connections {behavior_connection}* ]
  composite_declaration*

behavior_variable_decl ::=
  identifier : data_classifier_reference ;

behavior_state ::=
  behavior_defining_state_identifier:
  [ initial ] [ return | complete | urgent | composite ] state ;

behavior_state_transition ::=
  [behavior_transition_label : ]
  source_state_identifier { ,source_state_identifier}* -[ behavior_guard ]->
  destination_state_identifier { {behavior_action}* } ;

behavior_transition_label ::= identifier

behavior_guard ::=
  [ on behavior_expression --> ] behavior_event [ when behavior_expression ]
  | behavior_expression
  |

behavior_event ::=
  called_subprogram ! [behavior_parameter_bindings]
  | source_unique_port_identifier ? [behavior_parameter_bindings]

behavior_parameter_bindings ::=
  behavior_expression [ -> unique_port_identifier ]
  { , behavior_expression [ -> unique_port_identifier ] } *

behavior_connection ::=
  subprogram subprogram_feature_identifier -> subprogram_call_identifier ;
  | event [ data ] port unique_port_identifier -> subprogram_call_identifier ;
  | parameter unique_port_identifier -> unique_port_identifier ;

composite_declaration ::=
  composite history ? [ state state_idetifier | mode mode_idetifier ]
  states behavior_state*
  transitions behavior_state_transition*
  composite_declaration *
  end state_idetifier ;

behavior_expression ::= disjunction { or disjunction } *
disjunction ::= not_conjunction { and not_conjunction } *
not_conjunction ::= not ? conjunction
conjunction ::= arith_expression [ (<|<=|=|>|>=|!=) arith_expression ]
an_expression ::= add_expression { ( + | - ) add_expression } *

```

```
add_expression ::= basic_expression { ( * | / ) basic_expression } *

basic_expression ::=
  unsigned_aadlnumeric_or_constant
| behavior_variable_identifier
| loop_variable_identifier
| port_identifier
| port_identifier ' [ count | fresh ]
| data_access_identifier
| timeout behavior_expression
| data_subcomponent_identifier
| data_subcomponent_identifier [ behavior_expression ]
| data_access_identifier . data_subcomponent_identifier
| data_subcomponent_identifier . data_subcomponent_identifier
| data_classifier_reference . subprogram_identifier
  [ (behavior_expression{, behavior_expression}* ) ]
| data_classifier_reference . subprogram_identifier ?(behavior_expression)
| data_classifier_reference . subprogram_identifier ' parameter_identifier
  (behavior_expression)
| ( behavior_expression )

behavior_action ::=
  basic_action
| behavior_action behavior_action
| if ( behavior_expression ) behavior_action
  (elsif ( behavior_expression ) behavior_action)* ( else behavior_action )?
end if ;
| for ( identifier in integer_range ) { behavior_action }
| for ( identifier in data_classifier ) { behavior_action }

basic_action ::=
  computation ( behavior_expression , behavior_expression ) ;
| delay ( behavior_expression , behavior_expression ) ;
| communication ;
| assignment ;

assignment ::= behavior_expression := behavior_expression

communication ::= behavior_expression ( ? | ! ) behavior_parameter_bindings
```