AADL behavior annex

Jean-Paul Bodeveix\textsuperscript{2}  \quad Pierre Dissaux\textsuperscript{3}  \quad Mamoun Filali\textsuperscript{2}  \\
Pierre Gaufillet\textsuperscript{1}  \quad François Vernadat \textsuperscript{2}

\textsuperscript{1}AIRBUS-FRANCE  \\
\textsuperscript{2}FéRIA  \\
\textsuperscript{3}ELLIDIS

October 2006
Outline

1. Basic principles

2. The behavior annex
   - The expression language
   - The interaction language
     - Sending and receiving messages
     - Protocols
   - The action language
   - Interaction with the execution model

3. Conclusion and ongoing work
1. Basic principles

2. The behavior annex
   - The expression language
   - The interaction language
     - Sending and receiving messages
     - Protocols
   - The action language
   - Interaction with the execution model

3. Conclusion and ongoing work
Outline

1. Basic principles

2. The behavior annex
   - The expression language
   - The interaction language
     - Sending and receiving messages
     - Protocols
   - The action language
   - Interaction with the execution model

3. Conclusion and ongoing work
Main aspects

- Similar syntax to mode automata.
- Explicit scheduling of events.
- Assumes the AADL execution model with respect to: event ports, event data ports and server subprograms.
- Supported by the OSATE-TOPCASED environment.
Outline

1. Basic principles

2. The behavior annex
   - The expression language
   - The interaction language
     - Sending and receiving messages
     - Protocols
   - The action language
   - Interaction with the execution model

3. Conclusion and ongoing work
Contents of the annex

- Expression language.
- Interaction language.
- Action language.
Array types: currently they are declared as data subcomponents with a specific property which can be replaced by the current proposals. However, on the annex side, we have array expressions.

Union types: in the same way, they are declared as data subcomponents with a specific property (abstract). We have expressions for such a data type. These expressions are constructed through automatically generated constructors and destructors.
AADL side

- A **package** contains the subprograms: so called constructors for building values of the new data type.

- A **data** introduces the data type with the subprograms as constructors and a property: **Abstract** set to true specifies that the data is in fact an abstract data type with the subprogrames as its constructors.
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;
Annex side
Implicitly declared *observers* and *accessors* are specified:

- Observers allow to retrieve which constructor has been used to build the data.
- Accessors allow to retrieve which parameters have been used to build the data.
--- observers are named by adding 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?;

--- an accessor is defined for each parameter
--- of each constructor
--- this subprogram must only be called on a Failure data

subprogram Failure ’Reason
  features
    reason: out parameter Behavior::integer;
    this: requires data access Message
      {Required_Access => read_only;};
  end Failure ’Reason;
The interaction language

- State dependent scheduling of events.
- Explicit communication.
- Explicit protocols for server subprograms.
behavior annex proposal

\begin{verbatim}
annex behavior specification {**
  <state variables>?
  <initialization>?
  <states>?
  <transitions>?
  <connections>?
**}
\end{verbatim}
Sending and receiving messages

```plaintext
thread test
features
  p_in: in event data port Behaviour::integer;
  p_out: out event data port Behaviour::integer;
end test;
```
Sending and receiving messages

```plaintext
thread implementation test.default
subcomponents
  x: data Behaviour::integer;
annex Behaviour_behavior {**
  states
    s0: initial state;
    s1: state;
  transitions
    s0 −[p_in?(x)]→ s1 ;
    s1 −[p_out!(x+1)]→ s0 ;
**};
end test.default;
```
Example

shuffle

put1
ASER

put2
ASER

m
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 final state;
  s2: final state;
transitions
  s1 -> [put1?] s2;
  s2 -> [put2?] s1;
**};
end shuffle.i;
Protocols (HRT-HOOD)

Server_Call_Protocol:

\[
\text{type enumeration } (\text{ASER, HSER, LSER}) \Rightarrow \text{HSER applies to (server subprogram)};
\]

- ASER protocol: request deposit,
- LSER protocol: request deposit and acknowledgement wait,
- HSER protocol: request deposit and end processing wait (RPC).
The action language

- Basic actions.
- Sequences.
- Conditionals.
- Bounded loops.
<action> ::= 
   <basic_action> 
| <action> <action> 
| if ( <expression> ) <action> 
   (elsif (<expression>) <action>)* (else <action>) 
| for ( <identifier> in <integer_range> ) 
  { <action> } 
| for ( <identifier> in <data_classifier> ) 
  { <action> }
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;
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;
Outline

1. Basic principles

2. The behavior annex
   - The expression language
   - The interaction language
     - Sending and receiving messages
     - Protocols
   - The action language
   - Interaction with the execution model

3. Conclusion and ongoing work
Documentation (v. 1.5.1) available: project osate-ba
http://gforge.enseeiht.fr/projects/osate-ba

Integrated within the OSATE-TOPCASED environment.
small examples tested; plugin available at:
http://gforge.enseeiht.fr/projects/osate-ba

Experiment with a french space agency application.
semantics of the AADL execution model.
semantics of the annex and the interaction with the AADL execution model.