Cotre Annex
HRT-HOOD embedding

FéRIA

19th October 2005
Outline

1. Basic Principles
2. Data and types and subprograms
3. Message exchanges
4. HOOD objects
5. Temporal aspect
6. Cotre property set
7. Specification of system properties (preliminary study)
8. Specification of system properties (preliminary study)
   • Specific properties
   • Basic temporal operators
1. Basic Principles
2. Data and types and subprograms
3. Message exchanges
4. HOOD objects
5. Temporal aspect
6. Cotre property set
7. Specification of system properties (preliminary study)
8. Specification of system properties (preliminary study)
   - Specific properties
   - Basic temporal operators
1. Basic Principles
2. Data and types and subprograms
3. Message exchanges
4. HOOD objects
5. Temporal aspect
6. Cotre property set
7. Specification of system properties (preliminary study)
8. Specification of system properties (preliminary study)
   - Specific properties
   - Basic temporal operators
Outline

1. Basic Principles
2. Data and types and subprograms
3. Message exchanges
4. HOOD objects
5. Temporal aspect
6. Cotre property set
7. Specification of system properties (preliminary study)
8. Specific properties
   - Basic temporal operators
1. Basic Principles
2. Data and types and subprograms
3. Message exchanges
4. HOOD objects
5. Temporal aspect
6. Cotre property set
7. Specification of system properties (preliminary study)
8. Specific properties
9. Basic temporal operators
Outline

1. Basic Principles
2. Data and types and subprograms
3. Message exchanges
4. HOOD objects
5. Temporal aspect
6. Cotre property set
7. Specification of system properties (preliminary study)
8. Specification of system properties (preliminary study)
   - Specific properties
   - Basic temporal operators
Outline

1. Basic Principles
2. Data and types and subprograms
3. Message exchanges
4. HOOD objects
5. Temporal aspect
6. Cotre property set
7. Specification of system properties (preliminary study)
8. Specification of system properties (preliminary study)
   - Specific properties
   - Basic temporal operators
Embedding of HRT-HOOD concepts in AADL

- Property sets
- Behavioral annex (non deterministic specifications over AADL data)

Expression of system properties
Outline

1. Basic Principles
2. Data and types and subprograms
3. Message exchanges
4. HOOD objects
5. Temporal aspect
6. Cotre property set
7. Specification of system properties (preliminary study)
8. Specification of system properties (preliminary study)
   - Specific properties
   - Basic temporal operators
Simple and compound types

- Simple types: package with data declarations for integers, reals, ...
- Compound types (records): user defined hierarchies of data

```pascal
package Cotre
data integer
  properties Source_Data_Size => 32 bits;
end integer;

data float end float;

data boolean end boolean;
end Cotre;
```
Arrays

- No array type in AADL
- Reuse of UML multiplicity

```
tab: data Cotre::integer
    {CotreProperties::Multiplicity=>10;};
```

Multiplicity attribute

- Implicit declaration of access subprograms
- Usual array notation defined in Cotre behavioral annex
- Equivalent to new `data` declaration
package Cotre
    subprogram integer_10_put
        features parameter v: in Cotre::integer;
    end integer_10_put;

    subprogram integer_10_get
        features parameter v: out Cotre::integer;
    end integer_10_get;

    data integer_10
        features
            put: subprogram integer_10_put;
            get: subprogram integer_10_get;
    end integer_10;
Questions

- Should the array notation be extended to $N$ dimension arrays? The multiplicity attribute could take as value a list of $N$ integers.
- How to declare a symbolic constant for the array size?
Subprograms

- Behavior attached to subprogram implementations
- Access to subprogram parameters
- Access to visible data declared in AADL
Annex for subprogram behavior

- Specified by an automaton
- Reuse of mode automata syntax
- Action part associated to a transition
- Guard added to event
- \textit{return} state declaration: when reached,
  - output parameters are transmitted to caller,
  - control returns to caller.
Example (specification)

```
subprogram addition
features
  x: in parameter Cotre::integer;
  y: in parameter Cotre::integer;
  r: out parameter Cotre::integer;
  ovf: out parameter Cotre::integer;
end addition;
```
Example (implementation)

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

- AADL control flow: specification of unconditional call sequences
- proposed annex:
  - data dependant control flows
  - subprogram calls
  - raise of events
Example: subprogram calls in AADL

subprogram test end test;

subprogram implementation test.default
subcomponents
  x: data Cotre::integer;
  y: data Cotre::integer;
  z1: data Cotre::integer;
  z2: data Cotre::integer;
-- AADL : specification of call points
calls {
  add1: subprogram addition;
  add2: subprogram addition;
};
Example (2): AADL specification

```plaintext
-- fixed data flow graph
connections
parameter x -> add1.x;
parameter y -> add1.y;
parameter add1.z -> z1;
parameter z1 -> add2.x;
parameter y -> add2.y;
parameter add2.z2 -> z;
```
Example (3): Cotre specification

```
-- Cotre annex: dynamic sequencing of calls
annex cotre_behavior {**
  states
    s0: initial state;
    s1: state;
    s2: return state;
  transitions
    s0 -[addition!x,y,z1;]-> s1 {}
    s1 -[when z1 < 10 & addition!z1,y,z2;]-> s2 {}
    s1 -[when z1 >= 10]-> s2 { z2 := z1; }
**};
end test.default;
```
Raise of an event

subprogram addition
features
  x: in parameter std::integer;  y: in parameter std::integer;
  r: out parameter std::integer;  ovf: out event;
end addition;

subprogram implementation addition.default
annex cotre_behavior {**
states
  s0 : initial state;  s1 : return state;
transitions
  s0 -> s1 { r := x + y ; ovf := false; }  
  s0 -> s1 { }  
**}. 
The action part can contain subprogram calls

```plaintext
annex cotre_behavior {**
  states
  s0: initial state;
  s1: state;
  s2: return state;
  transitions
  s0 -> s1 { addition!x,y,z1; }
  s1 -[when z1 < 10] s2 { addition!z1,y,z2; }
  s1 -[when z1 >= 10] s2 { z2 := z1; }
**};
```
Outline

1. Basic Principles
2. Data and types and subprograms
3. **Message exchanges**
4. HOOD objects
5. Temporal aspect
6. Cotre property set
7. Specification of system properties (preliminary study)
8. Specification of system properties (preliminary study)

- Specific properties
- Basic temporal operators
use of ports and port groups of AADL
syntax similar to that of subprogram calls

- \( p!(t_1, \ldots, t_n) \) sends \( t_i \) to port group \( p \)
- \( p?(x_1, \ldots, x_n) \) receives \( x_i \) from port group \( p \)

use of ( ) to delimit port groups
Example of message transmission

```plaintext
thread test
features
  p_in: in event data port Cotre::integer;
  p_out: out event data port Cotre::integer;
end test;
```
Example of message transmission

```cotre
thread implementation test.default
subcomponents
  x: data Cotre::integer;
anex cotre_behavior {**
    states
      s0: initial state;
      s1: state;
    transitions
      s0 -[p_in?x]-> s1 {}
      s1 -[p_out!x+1]-> s0 {}
**};
end test.default;
```
Remark

- `p!` and `p?` can appear either in event or in action parts of transitions
- but different semantics ...
- in action part, execution *must* perform (wait for) the transmission
- in synchronization part, concurrence with other transitions
Outline

1. Basic Principles
2. Data and types and subprograms
3. Message exchanges
4. HOOD objects
5. Temporal aspect
6. Cotre property set
7. Specification of system properties (preliminary study)
8. Specification of system properties (preliminary study)
   - Specific properties
   - Basic temporal operators
Direct translation to AADL data declarations

```plaintext
subprogram put
  features v: in parameter Cotre::integer;
end put;
subprogram get
  features v: out parameter Cotre::integer;
end get;
subprogram empty
  features v: out parameter Cotre::boolean;
end empty;
subprogram full
  features v: out parameter Cotre::boolean;
end full;
```
data specification

data stack
features
  put: subprogram put;
  get: subprogram get;
  empty: subprogram empty;
  full: subprogram full;
end stack;
Subprograms may have a local behavior (annex of the subprogram implementation)

No access to data representation

Behavioral annex attached to the data component

- one entry for each subprogram (as specified by initial states)
- access to data implementation
- implicit call to local behavior of subprograms (on exit)
- the action part can call other subprograms
Basic Principles

Data and types and subprograms

Message exchanges

HOOD objects

Temporal aspect

Cotre property set

Specification of system properties (preliminary study)

Specication of system properties (preliminary study)

stack example

data implementation stack.default

elems: data Cotre::integer

annex cotre_behavior {

**

inits

sp := 0;

states

p : initial state for put;

g : initial state for get;

r : return state;

transitions

p - [when sp<5] -> s1
sp := sp + 1;
 elems[sp] := v;

q - [when sp>0] -> r
sp := sp - 1;
 elems[sp] := v;

f - [when sp = 5] -> r

v := (sp = 5);

e - [when sp = 0] -> r
v := (sp = 0);

}

end stack.default;

FéRIA
HOOD protected objects

- supported by AADL
  - data with property `Concurrenty_Control_Protocol`
- Access control left unspecified by AADL
  - property
    - `Supported_Concurrency_Control.Protocols` to be defined
Protected object example

data stack
features
  put: subprogram put;
  get: subprogram get;
  empty: subprogram empty;
  full: subprogram full;
properties
  Concurrency_Control_Protocol => protected
end stack;
HOOD active objects

- separation of processing and synchronization
- subprogram behavior specified in component annex
- object behavior: synchronization part
- AADL mechanisms weaker than HRT-HOOD:
  - asynchronous communications through ports,
  - highly synchronous communications through client/server subprograms,
  - no synchronization conditions
Asynchronous mode of HRT-HOOD

- Entry points are AADL ports
- Asynchronous message sending
- Thread associated to the server
- Bounded message queue (AADL attribute *Queue_Size*)
- Acceptance conditions specified by the server thread
Asynchronous communication example (1)

thread RW_monitor
features
    start_read : in event port {Queue_Size => 2};
    end_read   : in event port ;
    start_write: in event port ;
    end_write  : in event port ;
end RW_monitor;

thread implementation RW_monitor.i
subcomponents
    reading_cpt : data Cotre::integer;
    writing_flg : data Cotre::boolean;
Asynchronous communication example (2)

```plaintext
annex cotre_behavior {**
  inits
  reading_cpt := 0;  writing_flg := false;
  states
  s0 : initial state;
  transitions
  s0 -> s0 [when NOT writing_flg & start_read ?]
    { reading_cpt := reading_cpt + 1 };
  s0 -> s0 [end_read?]
    { reading_cpt := reading_cpt - 1 };
  s0 -> s0 [when !writing_flg & reading_cpt=0 & start_write?]
    { writing_flg := true };
  s0 -> s0 [end_write?]
    { writing_flg := false}
**}
```
Highly synchronous mode of HRT-HOOD

- The called service can return a result to the client.
- Partially implemented by AADL client/server subprogram
- Size of the server queue bounded (*Queue_Size* attribute)
- Activation conditions not specified in AADL
Proposed solution

- Use of the server thread to express conditions
- The server thread waits for allowed input events ...
- and implicitly calls the corresponding subprogram.
- The client waits for the completion of the called subprogram.

⇝ synchronization of three entities: the client, the server and the subprogram.
Highly synchronous communication example (buffer)

- **put** is allowed if buffer is not full
- **get** is allowed if buffer is not empty

```plaintext
thread serveur
features
  put: server subprogram buffer.put;
  get: server subprogram buffer.get;
end serveur;
```
Buffer behavior

```plaintext
thread implementation serveur.default
subcomponents
  buf: data buffer;
annex cotre_behavior {**
states
  empty: initial final state;
  full: final state;
  p: initial state for put; g: initial state for get;
  r: return state;
transitions
  empty -[put?v ]-> full { };
  full -[get?v ]-> empty { };
  p -[]-> r { buf.put!v }
  g -[]-> r { buf.get!v }
}
Remark (1)

Acceptance conditions could depend on entry point parameters

\[\Rightarrow\] more difficult to implement

\[\Rightarrow\] transmission must be undone

annex cotre_behavior {**
states
    full : state;
    empty : initial state;
transitions
    empty -[ put ?x & x > 0 ]-> full { };
    full -[ get ? ]-> empty { };
**}
Semi-synchronous mode of HRT-HOOD

- The client wakes up when message is taken into account
- No value is returned
- The server thread calls the subprogram associated to the entry point

\[ \text{specification of semi-synchronous mode: new property attached to entry point} \]

Server_Call_Protocol:

\[ \text{type enumeration (LSER, HSER) } \rightarrow \text{ HSER} \]
\[ \text{applies to (server subprogram);} \]
Asynchronous mode example

```
thread serveur 
features 
  put: server subprogram buffer.put; 
  {Server_Call_Protocol=>LSER;} 
end serveur; 
thread implementation serveur.default 
subcomponents 
  buf: data buffer; 
annex cotre_behavior {** states 
  s0 : initial final state; 
  p: initial state for put;  r: return state; 
transitions 
  s0 -[put?x]-> s0 { }; -- client wake up and subp call}
```
Multiple entry points - specification

thread serveur
features
    put_1: server subprogram put;  { Server_Call_Protocol => LSER }
    put_2: event data port Cotre::integer;  -- ASER
    get: server subprogram get;  { Server_Call_Protocol => HSER }
end serveur;
Multiple entry points - implementation I

```cpp
thread implementation serveur.default
subcomponents
  d: data Cotre::integer;
annex cotre_behavior {**
states
  busy : state; full : return state; empty : initial state;
p1: initial state for put_1; g: initial state for get;
r: return state;
```
transitions
empty [-[put_1? x]-> full \{d:=x;\}];
empty [-[put_2? x]-> busy \{}];
busy [-[]-> full \{d:=x;\}];
full [-[get?x]-> empty \{}];
p1 [-[]-> r \{d := x \}]
g [-[]-> r \{x := d; \}]
**}
end serveur.default;
Homogeneous entry point declarations

- Asynchronous entry points declared as ports
- Synchronous entry points declared as subprograms

⇝ only use subprogram syntax with ASER, LHSER or HSER mode
Example

```
put_2: event data port MyData; { ASER }
```

becomes

```
put_2: server subprogram put
  { Server_Call_Protocol => ASER }
```
Elapse of time

defined as new actions

- \text{Computation}(\text{min, max}): \text{non deterministic CPU usage}
- \text{Delay}(\text{min, max}): \text{non deterministic wait}
Periodic threads

Reuse of AADL properties attached to threads:

- Dispatch_Protocol=>Periodic
- Period=>...

The behavior defined by the behavioral annex starts from an initial state and must reach all final states before Compute_Deadline.
Example of periodic thread

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

thread implementation Producer.default
properties
  Dispatch_Protocol => Periodic;
  Period => 10 ms;
annexe cotre.behavior {**
  states
    s0: initial state;  s1: final state;
  transitions
    s0 -[ lput ! ]-> s1 {};
**}
```
Several possible declarations to bound synchronization waiting time.
The bound attribute may be attached to:

- subprogram declaration (the bound depends on the callee - HRT-HOOD)
- required subprogram declaration (the bound depends on the caller)
- subprogram effective call occurrence
Bounded synchronization time: proposal

- bound attached to interaction points
- guard syntax: \( g \text{ timeout } T: \text{true if } g \text{ is true since } T \text{ t.u.} \)
- the server computation cannot be interrupted
- \( \Rightarrow \) no bound for HSER synchronization mode
Bounded synchronization time: example

\[ s_0 \rightarrow [g \& p!] \rightarrow s_1 \{ \} \]
\[ s_0 \rightarrow [g \text{ timeout } T] \rightarrow s_2 \{ \} \]

- if synchronization on \( p \) occurs less the \( T \) t.u. after \( g \) becomes true, send message and go to \( s_1 \)
- else wait for \( T \) t.u. and go to \( s_2 \)
Outline

1. Basic Principles
2. Data and types and subprograms
3. Message exchanges
4. HOOD objects
5. Temporal aspect
6. Cotre property set
7. Specification of system properties (preliminary study)
8. Specification of system properties (preliminary study)
   - Specific properties
   - Basic temporal operators
**Cotre property set**

```plaintext
property set CotreProperties is
  Server_Call_Protocol: type enumeration (ASER, LSER, HSER)
              → HSER applies to (server subprogram);
  Multiplicity : aadlinteger applies to (data);
  Multiplicities : list of aadlinteger applies to (data)
end CotreProperties;
```
Outline

1. Basic Principles
2. Data and types and subprograms
3. Message exchanges
4. HOOD objects
5. Temporal aspect
6. Cotre property set
7. Specification of system properties (preliminary study)
8. Specification of system properties (preliminary study)
   - Specific properties
   - Basic temporal operators
Environment of a component

- Verification needs a closed system
- Compositional verification needs environment hypothesis
- attach an environment component to each component
- one to one correspondance between declared features
- Environment can be specified hierarchically
- Environment has a behavior
- Verification of the product (closed system)
Outline

1. Basic Principles
2. Data and types and subprograms
3. Message exchanges
4. HOOD objects
5. Temporal aspect
6. Cotre property set
7. Specification of system properties (preliminary study)
8. Specification of system properties (preliminary study)
   - Specific properties
   - Basic temporal operators
Environment of a component

- Verification needs a closed system
- Compositional verification needs environment hypothesis
  ⇝ attach an environment component to each component
  ⇝ one to one correspondance between declared features
- Environment can be specified hierarchically
- Environment has a behavior
- Verification of the product (closed system)
Specification of system properties

- branching time logic.
- state-based domain specific properties.
- event-based domain specific properties.
Re-initializable

- **resettable\_pot** (\(\equiv AG\ EF\ initial\))
  The component may return to its initial state (from any state).

- **resettable\_inev** (\(\equiv AG\ AF\ initial\))
  The component must return to its initial state (from any state).
Component liveness  Let $C$ be the subset of events of the component.

- **is_alive** $\equiv AG \ EF_C \ true$
  
  *Always some action of the component must be possible in the future*

  remark: When used in the root component, this property implies absence of deadlock.

- **no_livelock** $\equiv AG \ AF_C \ true$

  *A component may not be indefinitely idle (without performing some action)*
Invariant $p \ (\equiv AG \ p) \ Property \ p \ always \ holds$
Leadsto
\[ e_1 \text{ leadsto } e_2 \text{ [within } d \text{]} \equiv AG (e_1 \Rightarrow AF_{\leq d} e_2) \]
When expression \( e_1 \) holds then expression \( e_2 \) eventually holds in the future [at most in \( d \) unit time].

Reachable from
\[ \text{reachable } e_1 \text{ [from } e_2 \text{][within } d\] \equiv AG (e_1 \Rightarrow EF_{\leq d} e_2) \]
When expression \( e_1 \) holds then expression \( e_2 \) potentially holds in the future [at most in \( d \) unit time].

After
\[ e_1 \text{ after } e_2 \equiv A[\neg e_2 \text{ Weak Until } e_1] \]
When expression \( e_1 \) holds then expression \( e_2 \) has been necessarily satisfied in the past

Remark: Taking benefit of presence of labels of transitions it is also possible to envisage the use of behavioral equivalences