Modeling with the AADL

thomas.vergnaud@cnes.fr

Semantics of AADL Architectures

what kind of data?

```
data d
end d;

subprogram sp
features
  e : in parameter d;
  s : out parameter d;
end sp;
```

when are data actually sent? how many times?

```
thread node
features
  e : in event data port d;
  s : out event data port d;
end node;

thread implementation node.i
calls
  {call1 : subprogram sp;
   call2 : subprogram sp;};
connections
  parameter e -> call1.e;
  parameter e -> call2.e;
  parameter call1.s -> s;
  parameter call2.s -> s;
end node.i;
```

do both calls read the same value?

cnes

Current Situation

- as an ADL, the AADL itself only describe topologies
- lacks of clear semantics regarding described architectures
  - mainly provides a description of the thread lifecycles
  - does not specify when the communications occur
  - many ways to specify the data types
  - etc.
- behavior annex
  - behaviors within threads & subprograms
- programming language annex
  - translation from some AADL constructions to source code
- some elements on the wiki
  - semantics for data types

cnes

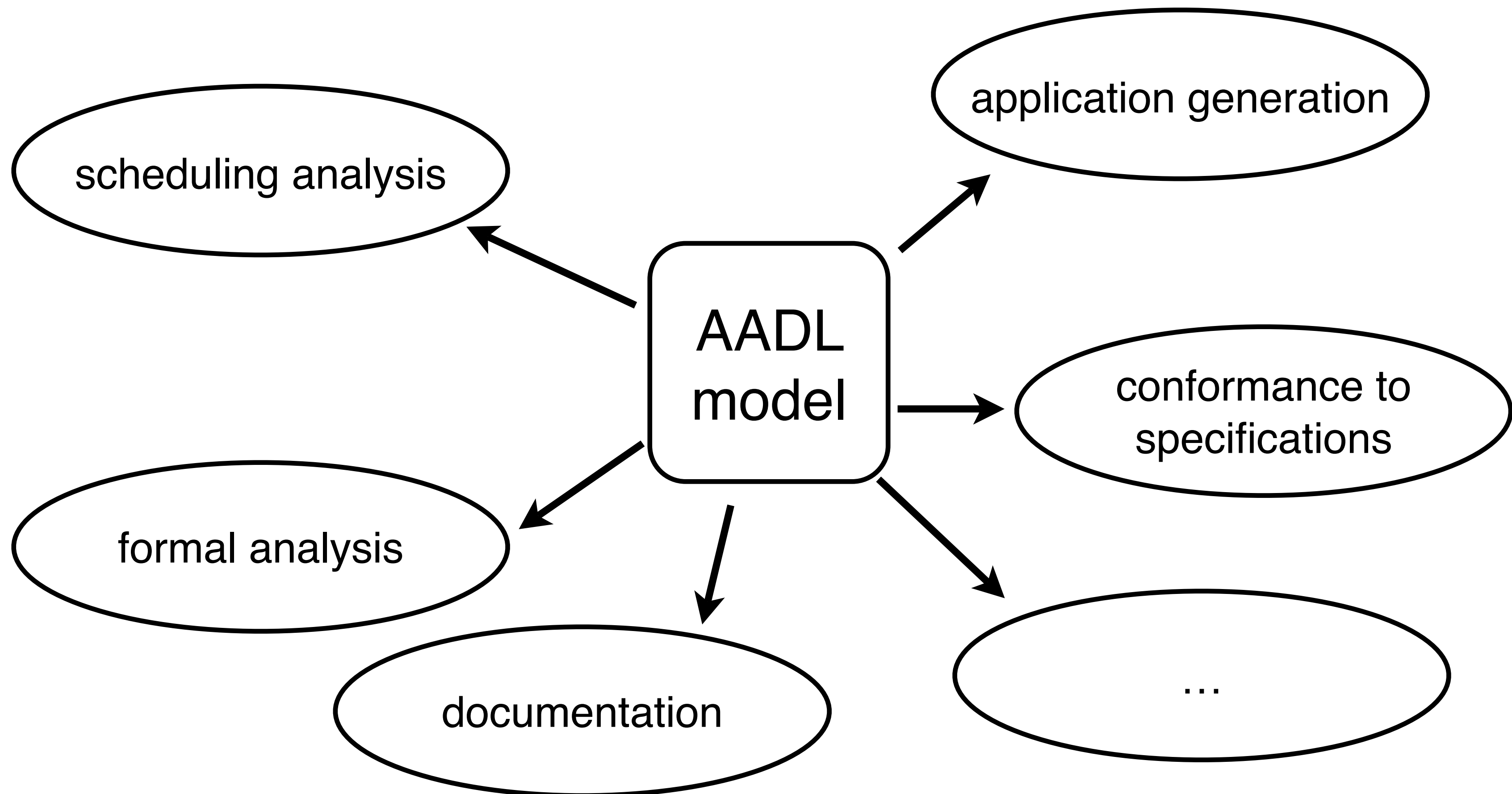
Scope of the Presentation

- patterns to help design architectures
- requirements for AADL runtimes
- semantics of AADL constructions
- overview of AADL annexes

cnes

Need for Common Architecture Semantics

- every tool must rely on the same assumptions



cnes

Semantics of Data Types

- use AADL properties
  - convenient for component extension
  - provide enough expressive power (simple types, combinations, data dimensions, etc.)
- already defined in the AADL wiki
- a pre-defined set of data components could be specified to ease the modeling activity

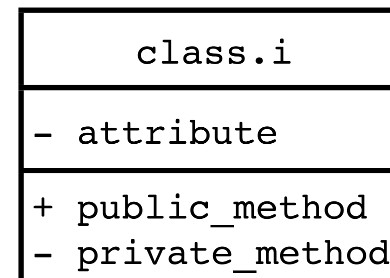
```
package Basic_Types
public
 data Integer
 properties
  data_semantics::data_type => integer;
 end Integer;
end Basic_Types
```

```
property set data_semantics is
data_type : enumeration
  (integer, float, boolean, string)
   applies to (data);
[…]
end data_semantics;
```

Architectural Patterns: Object-Oriented Architectures

- some architectures rely on object-oriented design
- classes are data components that provide methods
- objects are instances of classes
  = data subcomponents
- should we allow accesses to data? (= references to external data)
- some constructions do not (yet) conform to the AADL standard

```
data class
features
  public_method : subprogram;
end class;

data implementation class.i
subcomponents
  attribute : data;
  private_method : subprogram;
end class.i;
```



cnes

Architectural Patterns: Active and Passive Classes

- classes are modeled by data components
  - they represent passive components
- active objects should be modeled by threads
  - same construction as for passive classes
  - clear identification of the execution resources

Architectural Patterns: Communications Between Components

- communications between components require clear semantics
  - formal verification, application generation, scheduling analysis, etc. directly depend on this semantics
- to be specified by process & thread interfaces
  - processes & threads are the top-level application components

Architectural Patterns: Message Passing

- modeled by ports
  - event data ports : messages
  - event ports : messages with no data
  - data ports : messages without queues, and no triggering information

```
data d end d;

thread receiver_node
features
  e : in event data port d;
end receiver_node;

thread sender_node
features
  s : out event data port d;
end sender_node;
```

```
process proc end proc;

process implementation proc.i
subcomponents
  t1 : thread sender_node;
  t2 : thread receiver_node;
connections
  event data port t1.s -> t2.e;
end proc.i;
```

Architectural Patterns: Remote Procedure Calls

- provided/required subprogram accesses
  - required subprogram accesses are called from call sequences

```
subprogram sp end sp;

thread server_node
features
  rpc : provides subprogram access sp;
end server_node;

thread client_node
features
  rpc : requires subprogram access sp;
end client_node;
```

```
thread implementation client_node.i
calls
  {call1 : subprogram access rpc;};
end client_node.i;

process proc end proc;

process implementation proc.i
subcomponents
  t1 : thread client_node.i;
  t2 : thread server_node;
connections
  subprogram access t2.rpc -> t1.rpc;
end proc.i;
```

Architectural Patterns: Shared Memory

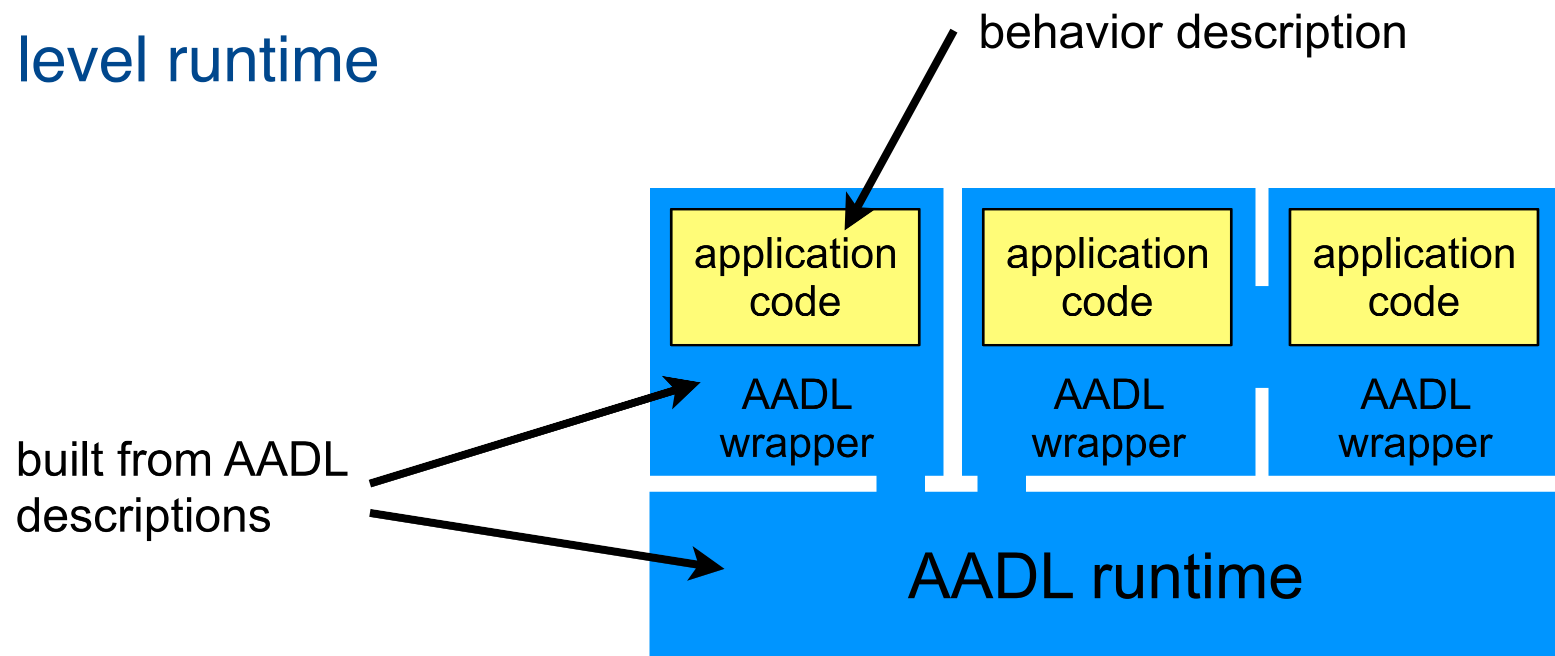
- required/provided data accesses
  - a data component is accessed by several threads
- some similarities with data ports, from an application point of view

```
data smem end smem;

thread application_node
features
  shared : requires data access smem;
end application_node;
```

```
process proc end proc;

process implementation proc.i
subcomponents
  t1 : thread application_node;
  t2 : thread application_node;
  d : data smem;
connections
  data access d -> t1.shared;
  data access d -> t2.shared;
end proc.i;
```

Defining a Runtime for AADL Applications

- AADL runtime = virtual machine
  = operating system + communication manager
- 2 approaches, depending on the required functionalities
  - high level runtime
  - low level runtime



cnes

High Level Runtime

- provide support for ALL the communication patterns
- applications rely on an abstraction layer
  - independent from the OS and hardware capabilities
- implicit resources to manage communications
- facilitates the modeling

cnes

Low Level Runtime

- does NOT manage communications
  - available communication mechanisms depend on the actual OS
  - restrictions on the modeling
- no implicit execution resources
- facilitates the precise evaluation of the architecture dimensions
- architectures for low level runtimes could be deduced from architectures for high level runtimes
  - by modeling high level runtimes in AADL

Execution Semantics: Call Sequences

- call sequences describe the *possible* execution flows in AADL threads or subprograms
- they are to be controlled by a behavior description
  - behavior annex
  - or source code
- if there is a unique call sequence, it is executed (implicit behavior)

cnes

Execution Semantics:

Data Subcomponents, Data Accesses & (Event) Data Ports (1)

- from the application point of view, data ports, subcomponents & accesses are variables
  - data subcomponents are local variables
  - required data accesses can be used to model global variables
  - (event) data ports point to I/O buffers

cnes

Execution Semantics:

Data Subcomponents, Data Accesses & (Event) Data Ports (2)

- data accesses, data subcomponents and (event) data ports are read and wrote according to subprogram connections
  - data that are connected to in parameters are read when the corresponding subprogram is called
  - same thing for writing operations
- data that are connected to (event) data ports
  - read data before executing the sequences
  - write data after executing the sequences

Execution Semantics: Example

- the first incoming data in `e` is stored in `var1`
- the second incoming data is passed to `call1`
- `call2` uses the first data, stored in `var1`
- `call3` uses the output of `call1`
- the output of `call1` is stored in `var2`
- the outputs of `call2` and `call3` are sent through `s`
- the content of `var2` is then sent through `s`

```
data d end d;

subprogram sp
features
  e : in parameter d;
  s : out parameter d;
end sp;

thread node
features
  e : in event data port d;
  s : out event data port d;
end node;
```

```
thread implementation node.i
subcomponents
  var1 : data d;
  var2 : data d;
calls
  {call1 : subprogram sp;
   call2 : subprogram sp;
   call3 : subprogram sp;};
connections
  cnx1 : event data port e -> var1;
  cnx2 : parameter e -> call1.e;
  cnx3 : parameter var1 -> call2.e;
  cnx4 : parameter call1.s -> call3.e;

  cnx5 : parameter call1.s -> var2;
  cnx6 : parameter call2.s -> s;
  cnx7 : parameter call3.s -> s;
  cnx8 : event data port var2 -> s;
end node.i;
```

Execution Semantics:
When Should Communication Occur?

- perform I/O before and after executing the threads
  - pass incoming data to the application
  - process the data
  - send output data
  - facilitates the analysis ; BUT there are issues with background threads
- perform I/O during the execution of the threads
  - more flexibility ; BUT analysis may be difficult
- let’s allow both
  - specified by an AADL property

Execution Semantics: Locking Access to Shared Data

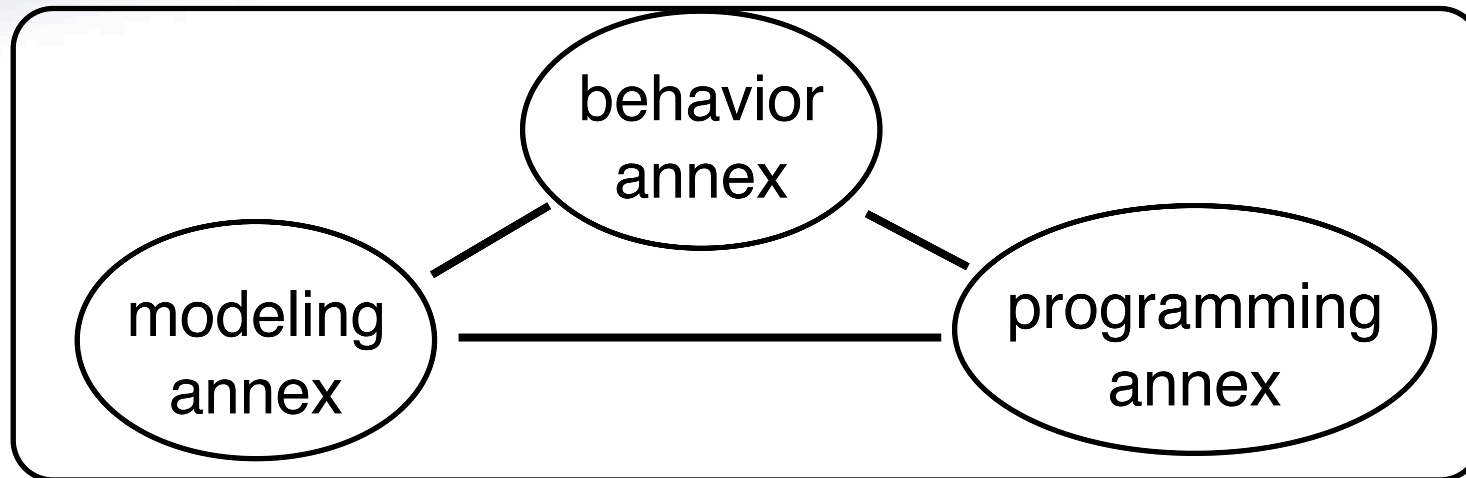
- threads may not systematically lock access to shared data
  - too restrictive
- need for a locking policy at subprogram level
  - specify that a subprogram call locks all data accesses (or not)
- finer grain policy should be achieved using the behavior annex or source code

cnes

API Provided by the Runtime

- data access lock
  - `get_resource`, `release_resource`
- execution
  - `await_dispatch`
  - sequence calls
- communications
  - `raise_event`, `send_data`, `get_data`, `read_data`, `write_data`

Toward Separation of Concerns



- modeling annex
  - semantics of AADL constructions
- programming language annex
  - standard runtime API
  - standard mappings for programming languages
- behavior annex
  - formal description of component behaviors

cnes

Specialized Property Sets

- instead of a monolithic property set
- it would help structure the modeling activities
  - specify what property sets should be involved in a given activity
    - scheduling analysis
    - deployment
    - code generation
    - etc.
  - facilitates the introduction of additional properties for extra activities.
    - modeling the power consumption of the architecture
    - connection with formal analysis
    - etc.

cnes

Conclusion

- the AADL itself only describes architecture topologies
- existing annexes
  - behavior, programming language
- need for a third annex
  - specify the semantics of AADL constructions
  - define the runtime services that can be used by behavior descriptions
  - would help in supporting a modeling methodology