AADL : about code generation

AADL objectives

- AADL requirements document (SAE ARD 5296)
  - Analysis **and** Generation of systems
- Generation can encompasses many dimensions
  1. Generation of skeletons from AADL components
     - Like from UML class diagrams
  2. Generation of system archetypes
     - Tasks, types, runtime configuration parameters, etc.
- In the following, we consider option #2
  - Supported by Ocarina, see http://www.openaadl.org

AADL and code generation

- AADL has a full execution semantics
  - Allow for full analysis:
    - Scheduling, security, error, behavior
- **Issue:** what about the implementation ?
  - How to go to code?
  - While preserving both the semantics and non functional properties ?
- **Solution:** enrich AADL with annexes documents
  - To describe application data
  - To detail how to bind code to AADL models

About AS5506/2 (Jan. 2011)

- *This document consists of three annexes to the SAE AADL standard that*
  - **The Data Modeling Annex** provides guidance on a standard way of associating data models expressed in other data modeling notations such as UML or ASN.1 with architecture models expressed in AADL,
  - **The Behavior Annex** enables modeling of component and component interaction behavior in a state-machine based annex sublanguage, and
  - **The ARINC653 Annex** provides guidance on a standard way of representing ARINC653 standard compliant partitioned embedded system architectures in AADL models.

About data modeling annex

- Allow one to clarify actual representation of data
  - Integer, floats, etc. with `Data_Representation`
- Actual size of data
  - 16/32/64 bits integers with `Source_Data_Size`
- Admissible range, precision
- Patterns for composite types, unions, etc.
- Based on a dedicated property set `Data_Model`

AADL: modeling data types

- **Solution:** enhance definition of types
  - One step closer to source code
  - Note: irrelevant for scheduling analysis

```
subprogram Receiver_Spg
features
  receiver_out : out parameter Target_Distance;
  receiver_in : in parameter Target_Distance;
end Receiver_Spg;

data Target_Distance
properties
  Data_Model::Data_Representation => integer;
end Target_Distance;
```

AADL and subprograms

- **Issue:** how to bind user code ?
- **Solution:** use default AADLv2 properties

```
subprogram Receiver_Spg
features
  receiver_out : out parameter Target_Distance;
  receiver_in : in parameter Target_Distance;
properties
  Source_Language => (Ada95); -- defined in AADL_Project
  Source_Name => "radar.receiver";
end Receiver_Spg;
```


AADL and programming languages

- **Issue:** how to map source code ?
- **Solution:** guidelines provided in the programming language annex document
  - Mapping rules from AADL and the target language
    - Similar to OMG IDL mappings for CORBA

```
subprogram Receiver_Spg
features
  receiver_out : out parameter Target_Distance;
  receiver_in : in parameter Target_Distance;
end Receiver_Spg;
```

```
procedure Receiver --  Ada
  (Receiver_Out : out Target_Distance;
   Receiver_In : Target_Distance);

void receiver /* C */
  (target_distance *receiver_out,
   target_distance receiver_in);
```

About AADL_Project

- AADL_Project is a property set, project specific
- Enumerators for particular configuration
- Defined w.r.t. model processing tools

```
Supported_Scheduling_Protocols: type enumeration
(SporadicServer, RMS, FixedTimeline, EDF, …

Supported_Concurrency_Control_Protocols: type enumeration
(None_Specified, Priority_Inheritance, Priority_Ceiling, ..

Supported_Source_Languages: type enumeration
(Ada95,C, Scade, Simulink, …
```


Attaching code to components

- Connecting subprograms to threads

```
thread receiver
features
  receiver_out : out data port radar_types::Target_Distance;
  receiver_in : in data port radar_types::Target_Distance;
end receiver;

thread implementation  receiver.impl
properties
  Dispatch_Protocol => Periodic;
  Compute_Entrypoint_Source_Text => « radar.transmitter » ;
  -- Attaching subprogram to thread, executed at each dispatch
end receiver.impl;
```

- Early specifications, for referring to a symbol

Attaching code to components

- Connecting subprograms to threads

```
thread receiver
features
  receiver_in : in event data port radar_types::Target_Distance
 {Compute_Entrypoint_Source_Text => « radar.transmitter » ;
  -- Attaching subprogram to port, executed at each dispatch
 };
end receiver;

thread receiver2
features
  receiver_in : in data port radar_types::Target_Distance
 {Compute_Entrypoint => classifier (transmitter_spg);
  -- Attaching subprogram to port, more precise
 };
end receiver2;
```

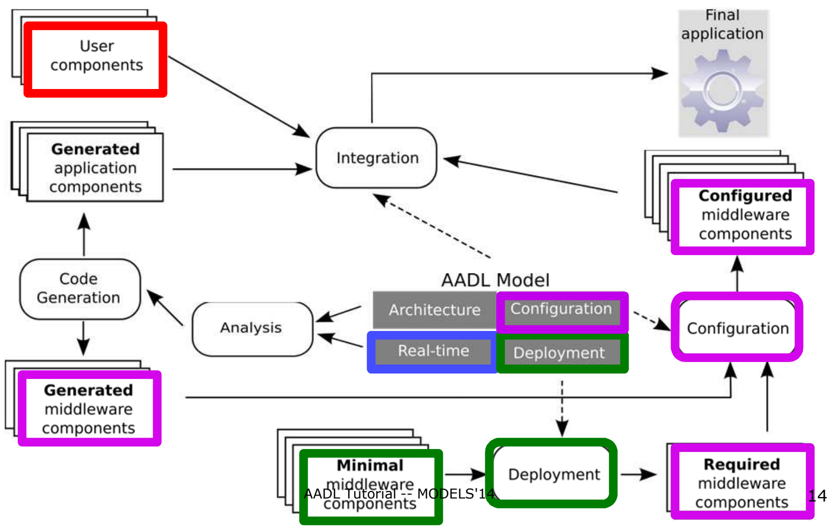

Attaching code to components

- Related properties
  - `Activate_Entrypoint:` upon thread activation
  - `Compute_Entrypoint:` dispatch
  - `Finalize_Entrypoint:` finalization
  - `Initialize_Entrypoint:` initialization of component
  - `Recover_Entrypoint:` in case of error
- Exist for both textual symbols (<x>_Source_Text property) or subprograms classifiers
- Applied to thread, device, subprogram, event port, event data port entities

AADL and code generation

- **Issue:** How much code should we write ? Tasks ? Queues ?
- **Answer:** the architecture says all
  - One can define a full framework and use it
    - Limited value
  - Generate as much things as possible
    - Reduce as much as possible error-prone and tedious tasks
- Ocarina: massive code generation
  - Take advantage of global knowledge to optimize code, and generate only what is required

Building process for HI-DRE systems using Ocarina



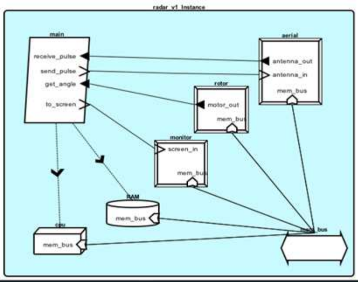
Benefits of code generation ?

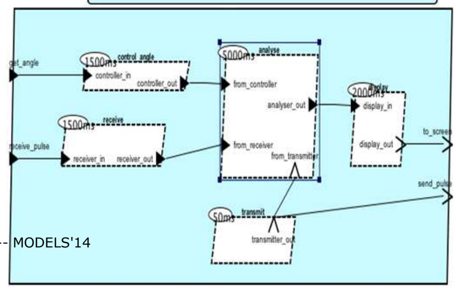
- **Is it worth a try ? Of course yes !**
- One pivot notation based on a unique notation
  - A-priori validation, using Cheddar, TINA ..
  - Optimized code generation
    - Measures show a difference of 6% in size
- Part of the promise of MBSE
  - One binary, no source code written for the most difficult part: the architecture, buffer, concurrency
  - Could be combined with other code generators like SCADE or Simulink to achieve zero-coding paradigm

Radar demo: code generation walkthrough

The Radar case study v1

- Model done with OSATE2
  - IMV for graphical view
- Text-based to have full control on properties
- Ocarina for code generation





Deployment on native target

- AADL Processor: execution platform

```
processor cpu_leon2
properties
   Scheduling_Protocol => (RMS);
   --  Configuration of scheduler
   Deployment::Execution_Platform => Native;
   --  Target platform
 end cpu_leon2;
```

- + simulation code for devices

Generating Cheddar + code

- Result from Cheddar

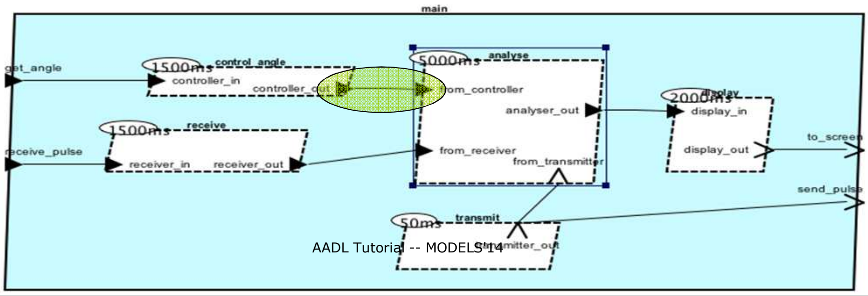
```
2) Feasibility  test based on
    worst case task response time :

- Bound on task response time :
    main_analyse => 130
    main_display => 70
    main_receive => 40
    main_control_angle => 20
    main_transmit => 10
-  All task deadlines will be met :
    the task set is schedulable.
```

- Traces from

```
macbookair-hugues% ./radar_v1/main/main
[ 0] Transmitter
[ 0] Controller, motor is at angular positio
[ 1] Analyser: target is at distance:  0 at
[ 1] Display_Panel: target is at ( 0, 0)
[ 1] Receiver, target is at distance 1
[ 500] Transmitter
[ 1001] Transmitter
[ 1500] Transmitter
[ 1500] Receiver, target is at distance 2
[ 1500] Controller, motor is at angular posi
[ 2000] Display_Panel: target is at ( 0, 0)
[ 2001] Transmitter
[ 2500] Transmitter
[ 3000] Transmitter
[ 3000] Receiver, target is at distance 3
[ 3000] Controller, motor is at angular posi
[ 3500] Transmitter
[ 4000] Transmitter
[ 4000] Display_Panel: target is at ( 0, 0)
```

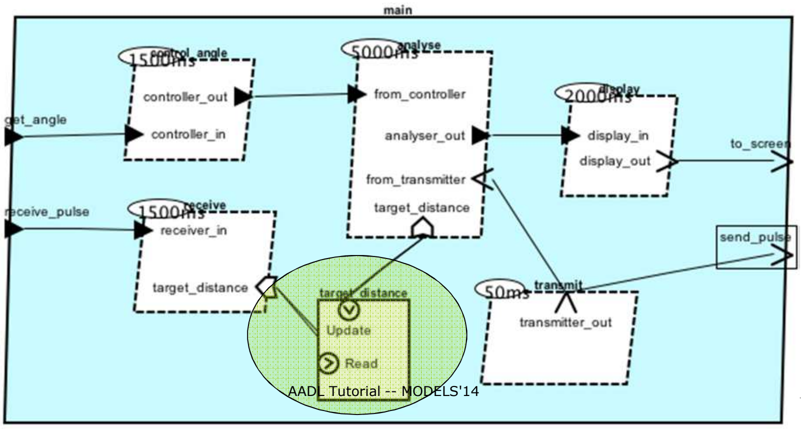

Assessment

- It works ;)
  - Execution traces meet scheduling simulation
  - And expected behavior
- Initial models use event ports
  - For each thread: one mutex + PCP is used



The Radar case study v2

- Change port communication with shared variable



AADL Tutorial -- MODELS'14 1

Generating Cheddar + code

- Result from Cheddar

```
2) Feasibility  test based on
    worst case task response time :

- Bound on task response time :
    main_analyse => 130
    main_display => 70
    main_receive => 40
    main_control_angle => 20
    main_transmit => 10
- All task deadlines will be met :
     the task set is schedulable.
```

- Traces from

```
macbookair-hugues% ./radar_v2/main/main
[ 0] Transmitter
[ 0] Controller, motor is at angular positio
[ 1] Analyser: target is at distance:  0 at
[ 1] Display_Panel: target is at ( 0, 0)
[ 1] Receiver, target is at distance 1
[ 500] Transmitter
[ 1001] Transmitter
[ 1500] Transmitter
[ 1500] Receiver, target is at distance 2
[ 1500] Controller, motor is at angular posi
[ 2000] Display_Panel: target is at ( 0, 0)
[ 2001] Transmitter
[ 2500] Transmitter
[ 3000] Transmitter
[ 3000] Receiver, target is at distance 3
[ 3000] Controller, motor is at angular posi
[ 3500] Transmitter
[ 4000] Transmitter
[ 4000] Display_Panel: target is at ( 0, 0)
```

AADL Tuto

Assessment

- It still works ;)
- We can exploit models a little more

```
data PO_Target_Distance
 features
  --  …
 properties
   Concurrency_Control_Protocol => Priority_Ceiling;
   --  Priority is not set, will use default value
   --  of maximum priority
 end PO_Target_Distance;
```

- Cheddar indicates that Scheduling simulation, processor cpu :
  - Number of preemptions : 0
  - Number of context switches : 4
- We can change protocol to none safely

AADL & other MDE frameworks

Integration with Simulink, SCADE et al.

AADL and other modeling notations

- AADL helps modeling **architectures**
  - Capture key aspects of design: hardware/software
  - Expression of some non functional properties: priority, resource consumption, latency, jitter, …
  - Enables: scheduling analysis, resource dimensioning, mapping to formal methods, fault analysis, …
- Functional notations (Simulink, SCADE, ..) describes precisely system behavior
  - Provides a high-level behavioral/computational view
  - mapped onto hardware/software elements
- Natural complement to ADLs

"Zero coding" paradigm

- Code generation from models is now a reality
  - Proposed by many tools
- Functional models
  - kcg: SCADE's certified code generation
  - Simulink Coder
- Architectural models
  - Ocarina: AADL code generator for HIsystems
- Foundations for a "zero coding" approach
  - Model, then integrate code generated from each view
- **Issue:** which integration process ?
  - Two approaches, driven by user demand

Code generation patterns

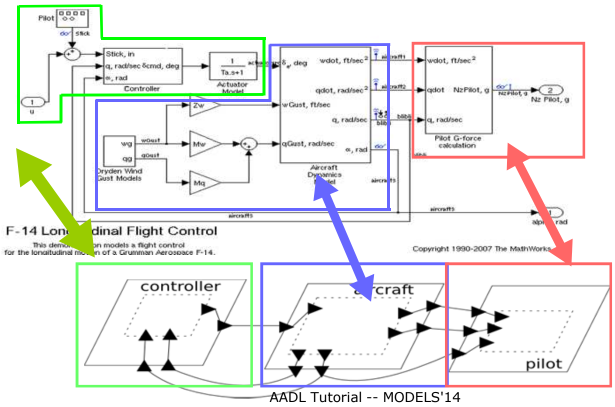
- Each functional framework relies on same foundations
  - Synchronous: discrete computation cycles
  - Asynchronous: function calls
- SCADE/Simulink/Esterel: a 3-step process
  - Fetch **in** parameters from AADL subprograms
  - Call the **reaction function** to compute output values
  - Send the output as **out** parameters of the AADL subprogram
- Architectural blocks are mapped onto programming language equivalent constructs
  - Ocarina relies on stringent coding guidelines to meet requirements for High-Integrity systems, validated though test harness by ESA, Thales, SEI, and their partners

From AADL + X tocode

- Ocarina handles all code integration aspects
  - How to map AADL concepts to source code artefacts (POSIX threads, Ada tasks, mutexes, ...)
  - Handle portability concerns to several platforms, from bare to native
- + some knowledge on how a SCADE or Simulink models is mapped onto C code
  - So that integration is done by the code generator
  - No manual intervention required
- Supports **"zero coding"** approach

Application-driven process

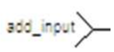
- Functions may be defined first, then refined to be bound to an existing architecture”

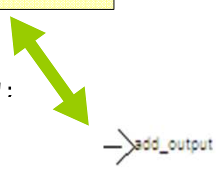


Architecture-driven process

- Reverse option: architecture is defined first, then a skeleton of the functional model is deduced, then implemented

```
subprogram spg_scade
features
input: in parameter integer {Source_Name => "add_input";};
output: out parameter integer {Source_Name => "add_output";};
properties
        source_name => "inc";
        source_language => Scade;
        source_location => "/path/to/scade-code/";
end spg_scade;
```

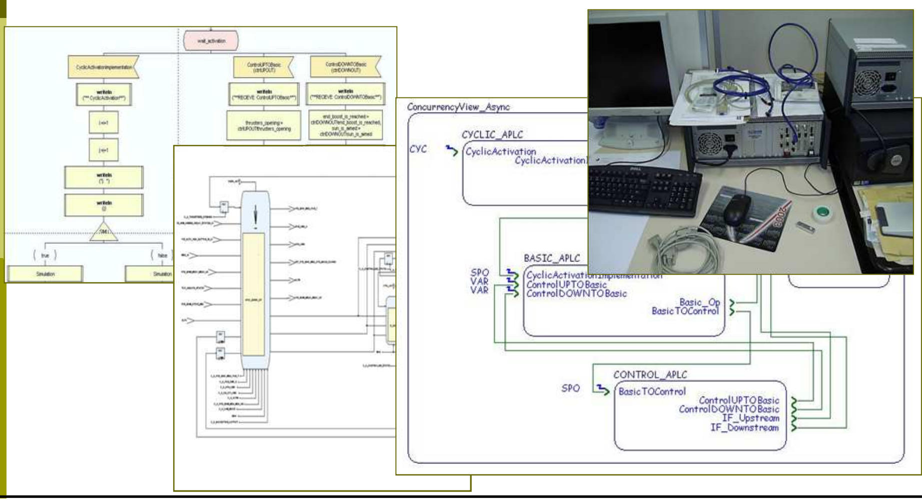




How to bind to AADL models ?

- In both cases, we rely on standard AADLv2 patterns
  - Source_Language <-> SCADE or Simulink
  - Source_Name <-> SCADE node or Simulink block
  - Source_Location <-> path to kcg orSimulink Coder generated code
- Smooth integration of AADL and other functional modeling
  - Providing only required information
  - While remaining 100% automatic

TASTE: DSML as inputs, AADL at its core



Conclusion

- System are heterogeneous, so are models
  - AADL separates architecture from functional models
  - Allows reference from the architecture to function blocks
- Integration of AADL and SCADE or Simulink in to perform full generation of systems is desirable
- Advantages
  - "Zero coding" paradigm to ease integration work
  - Quality of code generated for both functions and architecture
  - Opens the path towards qualification/certification of complex embedded systems at model-level