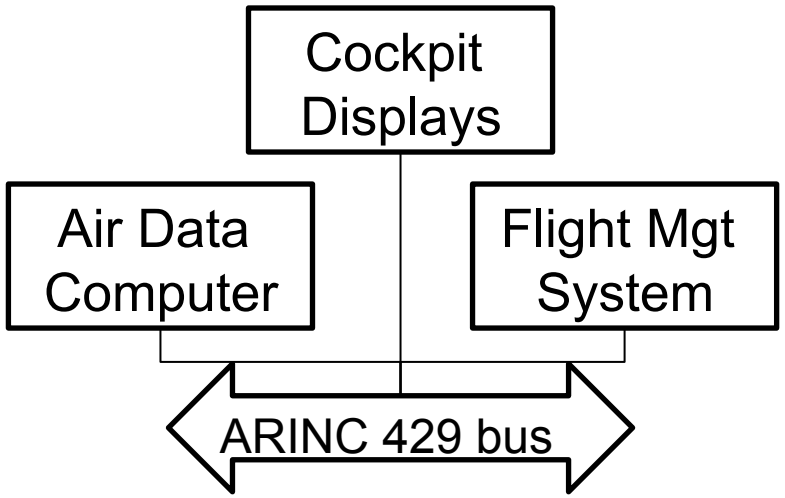
ARINC653 annex

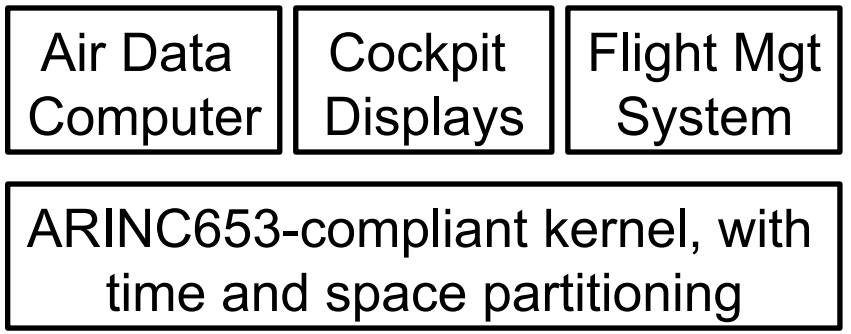
About AADL annexes

- ***AS5506/2 (January 2011)***
  - **Data Modeling Annex** provides guidance on a standard way of associating data models expressed in other data modeling notations (C or ASN.1) with architecture models expressed in AADL,
  - **Behavior Annex** enables modeling of component and component interaction behavior in a state-machine based annex sublanguage,
  - **ARINC653 Annex** provides guidance on a standard way of representing ARINC653 standard compliant partitioned embedded system architectures in AADL models.
- ***AS5506/1A (October 2015 ?)***
  - **Code generation Annex** defines language-specific rules for source text to be compliant with an architecture specification written in AADL;
  - **Error Model Annex** defines features to enable the specification of redundancy management and risk mitigation methods in an architecture, and enable qualitative and quantitative assessments of system properties such as safety, reliability, integrity, availability, and maintainability.

About the ARINC653 standard

- **ARINC653 aims at supporting the Integrating Modular Avionics conceptual framework, evolution of federated architecture**
  - » Multiple functions are allocated on the same processor with space and time isolation
- **ARINC653 defines a set of software API for Safety-Critical avionics Real-time Operating Systems**





ARINC653 APEX

- **APEX, the Application EXecutive API services:**
  - Process, time, partition, sampling and queuing ports,
  - Buffer, blackboard, semaphore, event error management
  - Health management
  - With a C and Ada API

- **The APEX is meant for modularity, portability**
  - A few system calls, 53
  - Configuration through external XML files to reduce code/ configuration coupling, delegation of responsibilities

Rationale for the ARINC653 Annex

- **AADLv2 added concepts to support IMA systems**
- **Most important being the notion of "virtual processor"**
  - A dedicated scheduling and memory space inside a processor
- **Yet, many issues were still open**
  - How to model blackboards, semaphores?
  - How to represent fault protection mechanism
  - How to take into account variability in APEX implementations?
- **Goal of the ARINC653 Annex document**
  - Define modeling patterns for IMA systems
  - Define additional property set when needed to clarify concepts
- **Based on ARINC653-2, published in 2006**

Mapping ARINC653 concepts

- **ARINC module = AADLv2 processor**
  - AADL processor defines the OS + configuration parameters
  - Additional properties for major frame, slot allocation, etc.
- **ARINC partitions = AAVLv2 virtual processor**
  - Link to partitions as virtual processors

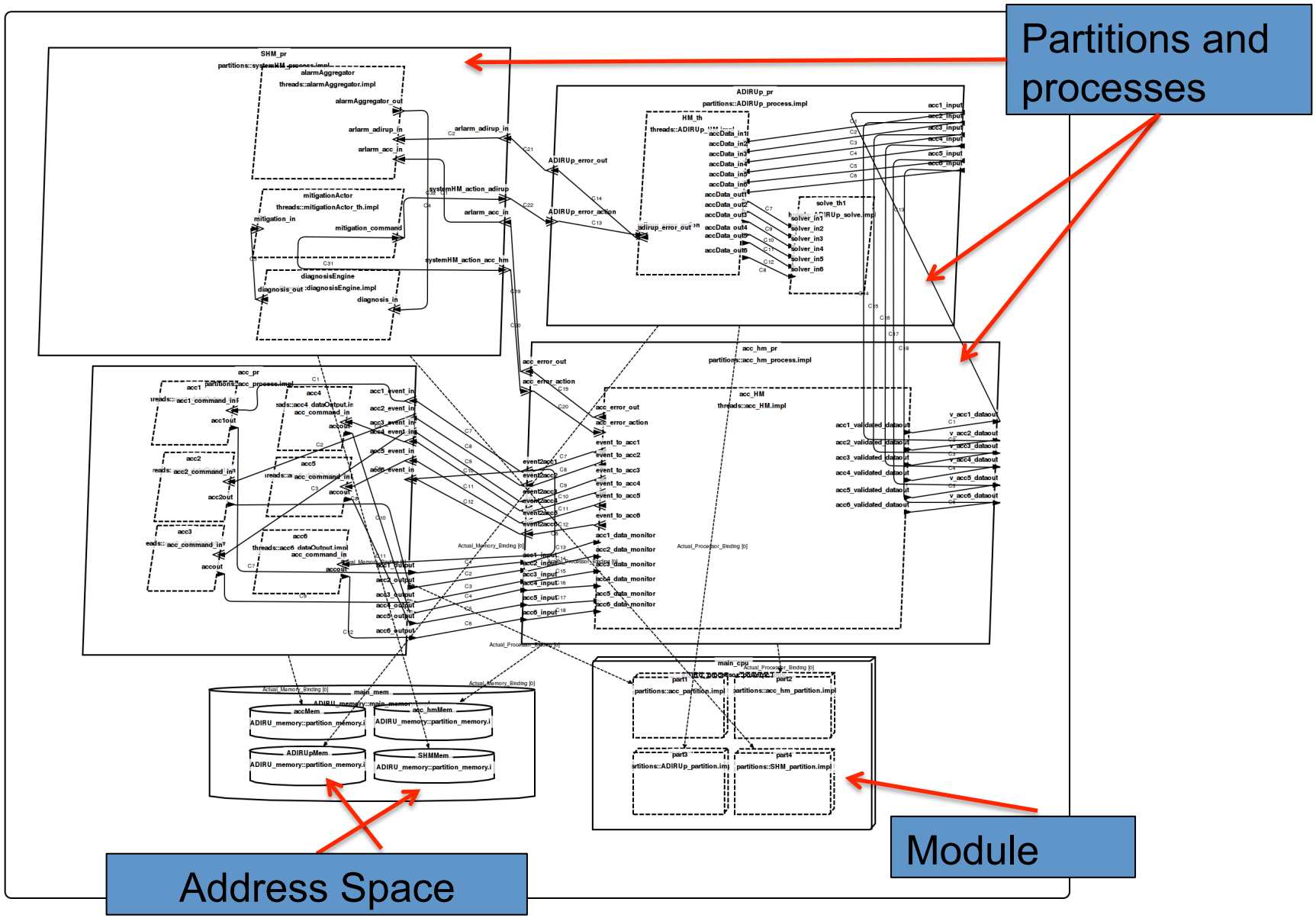
```
virtual processor implementation acc_partition.impl
end acc_partition.impl;

processor powerpc end powerpc;

processor implementation powerpc.impl
subcomponents
  --  ARINC653 partitions are subcomponents of the PowerPC component
  part1: virtual processor partitions::acc_partition.impl
              { ARINC653::Partition_Identifier => 1;
                ARINC653::Partition_Name       => "acc"; };
properties
  ARINC653::Module_Major_Frame => 150 ms;
  ARINC653::Module_Schedule =>
            ( [Partition => reference (part1); Duration  => 1 ms;
               Periodic_Processing_Start => true;] );
```

Mapping ARINC653 concepts, cont'd

Other mappings is semantic adaptation of concepts

- » ARINC 653 process = AADLv2 thread
  - Rationale: ARINC653 process are OS thread
- » ARINC653 queuing ports = AADLv2 event (data) ports
- » ARINC653 sampling ports = AADLv2 data ports
  - Semantics are similar, with equivalent configuration parameters for queue size, refresh period, etc.
- » ARINC653 buffers = AADLv2 event data ports
- » ARINC653 blackboard = AADLv2 data port or data components
  - To model inter-process communication in the same address space

ADIRU, graphical representation



AADL and XML configuration data

- **ARINC653 Executives require an additional configuration file**
- **A (full) AADL model must define all components**
  - For analysis or code generation purposes

- **Can derive configuration file from the AADL model**
  - Implemented in Ocarina, targets DeOS and VxWork653
- **Part of the model bus philosophy**
  - One repository that can be mined for various purposes
    - Analysis, code generation, management of configuration parameters

AADL and code generation

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

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

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

```
data C_Unsigned_Long_Int
 --  This data component defines a C unsigned long int type, with a
 --  dual nature The first properties defines its representation in
 --  memory, the two last its mapping in C.
properties
 Data_Model::Data_Representation => integer;
 Data_Model::Number_Representation => unsigned;
 Data_Size => 4 bytes;
 Source_Language => (C);
 Type_Source_Name => "unsigned long int";
end C_Unsigned_Long_Int;

data accData extends C_Unsigned_Long_Int
end accData;

subprogram acc1_dataOutput_spg
features
   acc1DataOut: out parameter SHM_DataType::accData;
   event_in:   in parameter SHM_DataType::actionData;
end acc1_dataOutput_spg;
```

AADL and subprograms

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

```
subprogram acc1_dataOutput_spg
features
  acc1DataOut: out parameter SHM_DataType::accData;
  event_in:    in parameter SHM_DataType::actionData;
properties
  Source_Language => (C);
  Source_Name =>"acc1dataoutput";
  Source_Text => ("../../../acc_code.o");
end acc1_dataOutput_spg;
```

AADL and programming languages

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

```
subprogram acc1_dataOutput_spg
features
  acc1DataOut: out parameter SHM_DataType::accData;
  event_in:    in parameter SHM_DataType::actionData;
end acc1_dataOutput_spg;
```

```
procedure acc1_dataOutput_spg ( -- Ada
 (acc1DataOut: out SHM_DataType.accData;
  event_in:    in  SHM_DataType::actionData);

void acc1_dataOutput_spg ( /* C */
(acc1DataOut *SHM_DataType_accData,
 event_in:   SHM_DataType_actionData);
```

Attaching code to components

- **Connecting subprograms to threads**
  - » Connect ports to parameters
  - » Use call sequence attached to thread

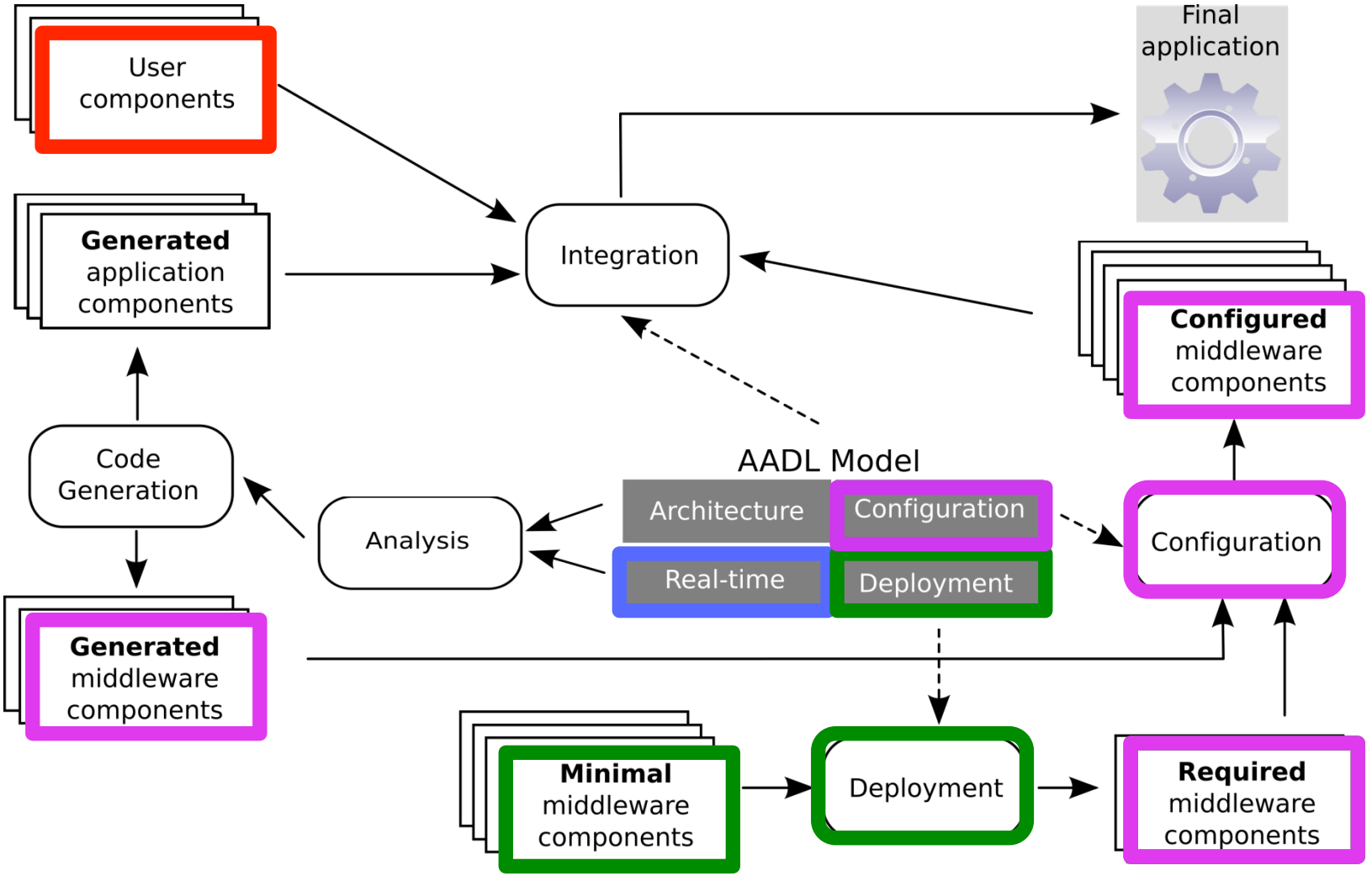
```
thread acc1_dataOutput
features
  acc1out: out data port SHM_DataType::accData;
  acc1_command_in: in event data port SHM_DataType::actionData;
properties
  Dispatch_Protocol => Periodic;
  -- …
end acc1_dataOutput;

thread implementation  acc1_dataOutput.impl
calls
  sub1: { spg: subprogram subprograms::acc1_dataOutput_spg;};
connections
  C1: parameter spg.acc1DataOut->acc1out;
  C2: parameter acc1_command_in->spg.event_in;
end  acc1_dataOutput.impl;
```

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
  - Support for regular RTOS (POSIX, Xenomai, FreeRTOS) and ARINC653 APEX (DDC-I DeOS and WRS VxWorks653)

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

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

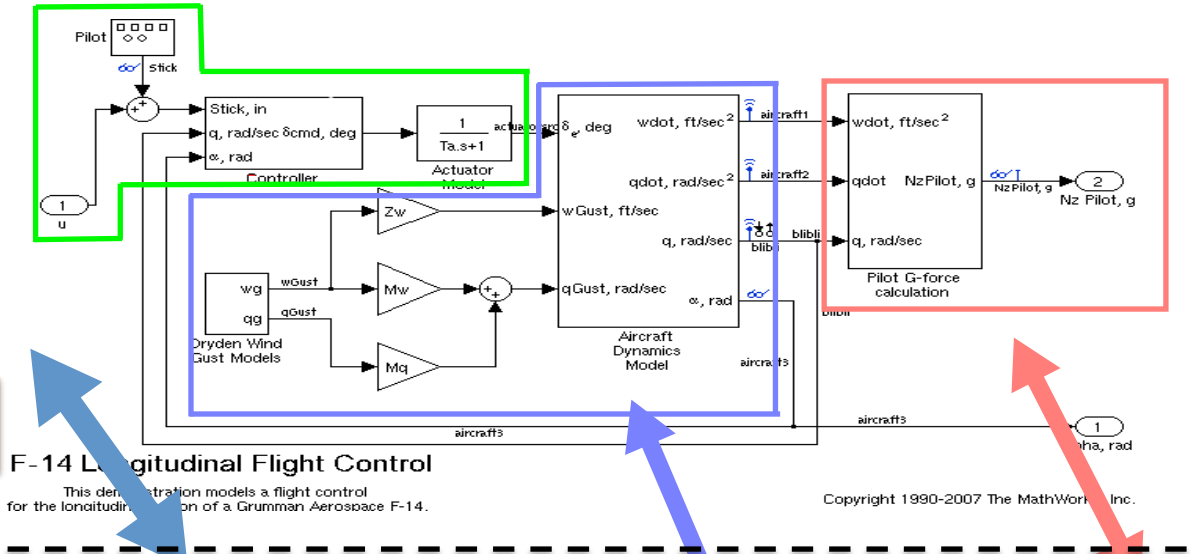
From AADL + X tocode

- **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**

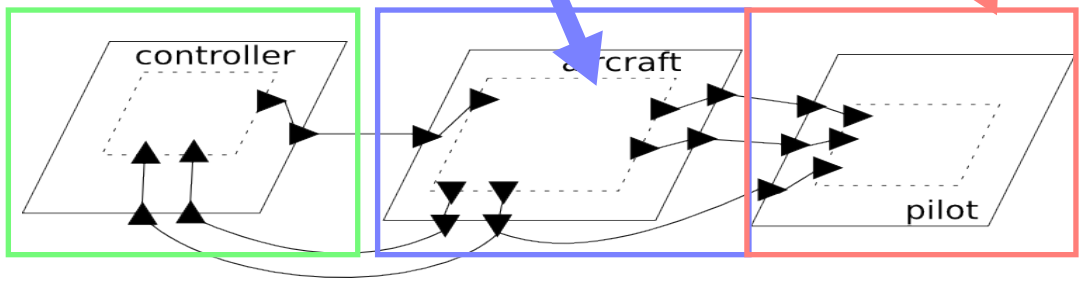
Application-driven process

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

SCADE/Simulink



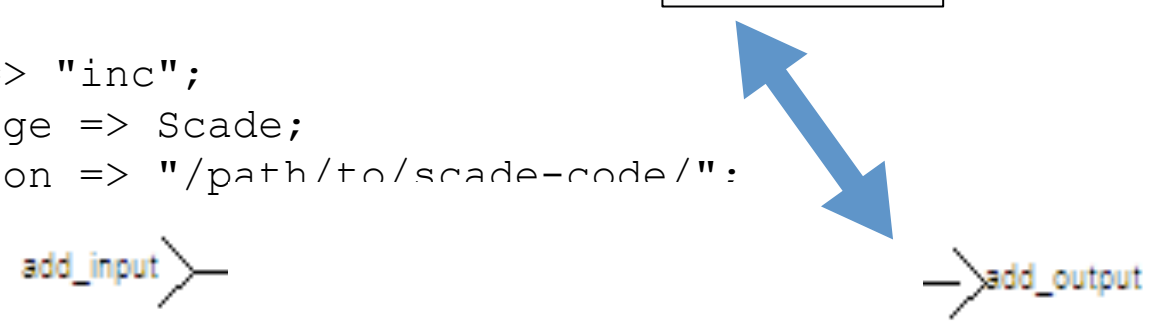
AADL



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;
```

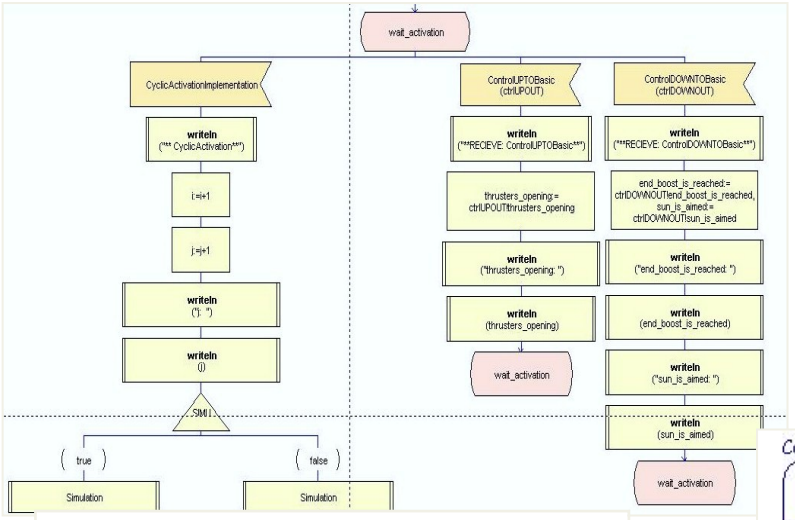


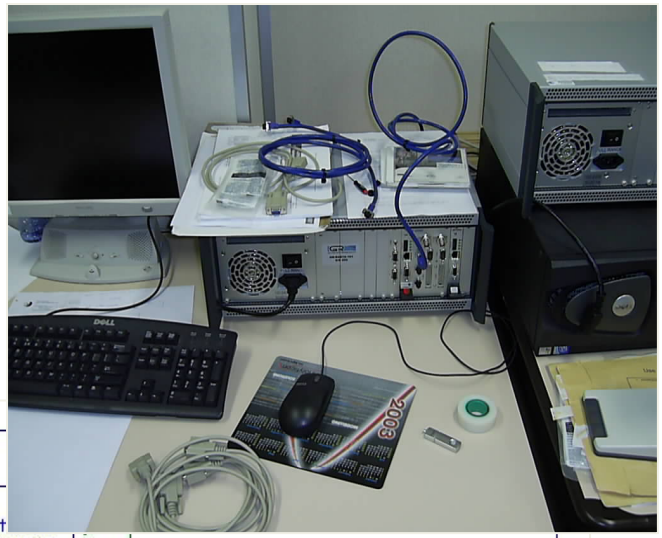
How to bind to AADL models ?

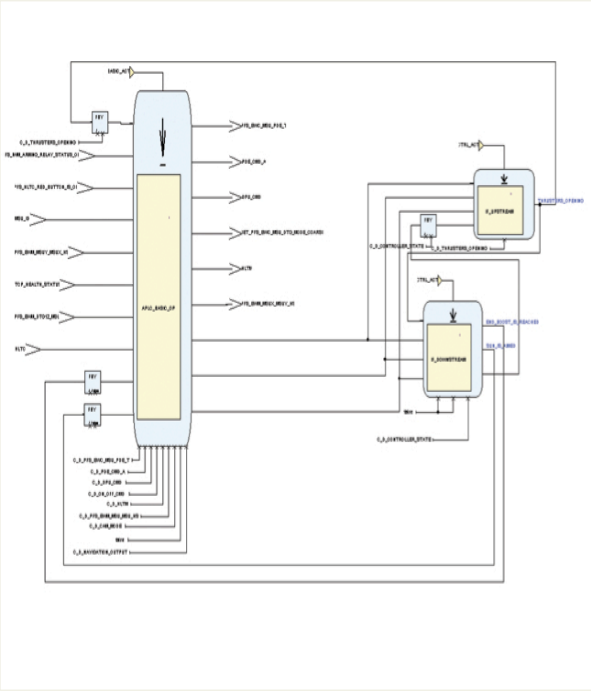
- **In both cases, we rely on standard AADLv2 patterns**
  - » Source_Language <-> SCADE or Simulink
  - » Source_Name <-> SCADE node or Simulink block
  - » Source_Location <-> path to kcg orSimulink Coder generated code

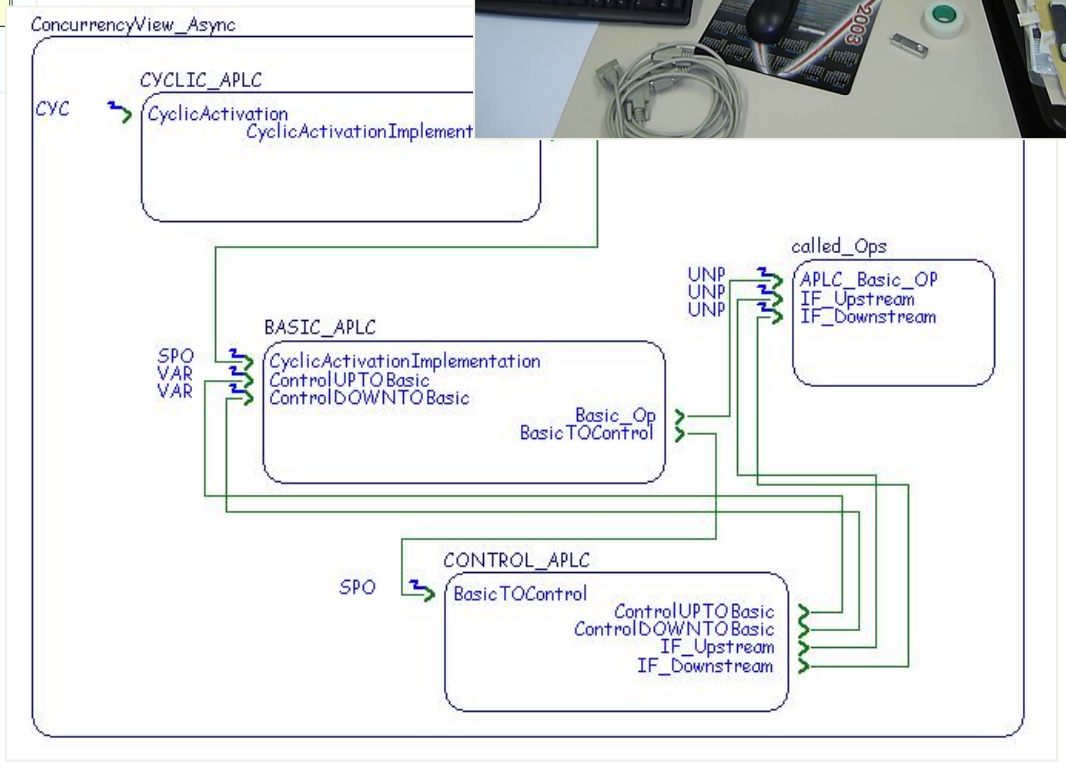
- **Smooth integration of AADL and other functional modeling**
  - » Providing only required information
  - » While remaining 100% automatic

TASTE: DSML as inputs, AADL at its core









SCADE integration example

- **Integrate SCADE on ARINC653 systems**
  - Software behavior captured with SCADE
  - Architecture specified with AADL
- **Auto-Generate Architecture and "glue code"**
  - Generate ARINC653 configuration and partitions code
  - Different OS, same behavior
- **No need for manual code**
  - Smooth and integrated process

http://aadl.info/aadl/demo-arinc653/

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 to perform full generation of systems proved to be effective**
- **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