

# Scheduling analysis of AADL architecture models

**Frank Singhoff+, Pierre Dissaux\***

**+Lab-STICC/CNRS UMR 6285, Université de Bretagne  
Occidentale, France**

**\*Ellidiss Technologies, France**



# Outline

---

**Goal:** overview of scheduling analysis capabilities that are proposed by the AADL and tools implementing it. Show the benefits that can be expected by performing early scheduling analysis for real-time software.

- ❑ **Part 1: introduction to AADLv2 core (about 2h/2h30)**
  - Syntax, semantics of the language
- ❑ **Part 2: introducing a case study (about 15')**
  - A radar illustrative case study
- ❑ **Part 3: scheduling analysis (about 2h/2h30)**
  - Introducing real-time scheduling and its use with AADL
- ❑ **Part 4: practical labs, exercises, discussion (about 1 or 2 hours)**
  - How to use tools in order to apply what we learnt in parts 1 to 3

# CPS-WEEK Agenda

---

- 9:00-10:00 tutorial
- 10:00-10:30 coffee break
- 10:30-12:30 tutorial
- 12:00-13:30 lunch break
- 14:00-15:00 tutorial
- 15:00-15:30 coffee break
- 15:30-17:30 tutorial

# Acknowledgments

---

- Many of those slides were written with or by Jérôme Hugues/ISAE, for the following tutorials:
  - AADLv2, An Architecture Description Language for the Analysis and Generation of Embedded Systems. J. Hugues, F. Singhoff. Half day tutorial presented in the ACM HILT conference, Portland, USA, October 2014.
  - AADLv2, a Domain Specific Language for the Modeling, the Analysis and the Generation of Real-Time Embedded Systems. F. Singhoff, J. Hugues. Half day tutorial presented in the International MODELS conferences, Valencia, Spain, September 2014.
  - AADLv2, an Architecture Description Language for the Analysis and Generation of Embedded Systems. J. Hugues F. Singhoff. Half day tutorial presented in the International EMSOFT/ESWEEK conferences, Montreal, Canada, September 2013.
  - Développement de systèmes à l'aide d'AADL - Ocarina/Cheddar. J. Hugues, F. Singhoff. Tutoriel présenté à l'école d'été temps réel (ETR'2009). Septembre 2009. Pages 25-34. Paris.
- Thank you Jérôme :-)

# We focus on Real-Time, Critical, Embedded Systems

---

- « *The correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced* »  
Stankovic, 1988.
  
- **Properties we look for:**
  - Functions must be predictable: the same data input will produce the same data output.
  - Timing behavior must be predictable: must meet temporal constraints (e.g. deadline).

# We focus on Real-Time, Critical, Embedded Systems

---

- ❑ **Critical real-time systems:** temporal constraints **MUST** be met, otherwise defects could have a dramatic impact on human life, on the environment, on the system,
- ❑ **Embedded systems:** computing system designed for specific control functions within a larger system.
  - Often with temporal constraints.
  - Part of a complete device, often including hardware and mechanical parts
  - Limited amount of resources.

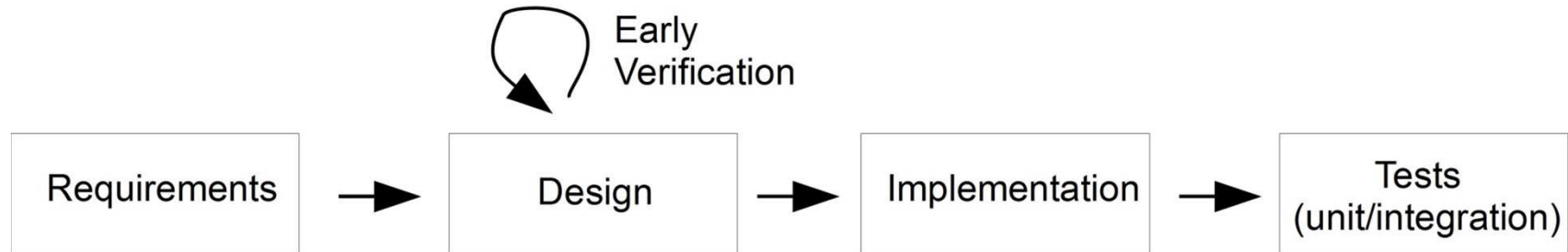
# We focus on Real-Time, Critical, Embedded Systems

---

- **Examples:** aircraft, satellite, automotive, ...
- 1. Need to handle time. Concurrent applications.
- 2. May have dramatic impact on human life, on the system, ...
- 3. Do not allow software maintenance => difficult to correct erroneous software/bugs.
- 4. High implementation cost : temporal constraints verification, safety, dedicated hardware/software

# We focus on Real-Time, Critical, Embedded Systems

---



- ❑ **Specific software engineering** methods/models/tools to master quality and cost
  - Example : early verifications at design step




# Motivation for early verification

## □ From NIST 2012:

- 70% of fault are introduced during the design step ; Only 3% are found/solved. Cost : x1
- Unit test step: 20% of fault are introduced ; 16% are found/solved. Cost : x5
- Integration test step: 10% of fault are introduced ; 50% are found/solved. Cost : x16

□ **Objective:** increase the number of faults found at design step!

□ **Early verification:** multiple verifications, including expected performances, e.g. can deadlines be met?



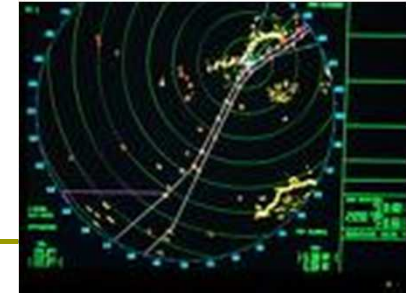
Mission Systems Architecture  
Challenges for Future Vertical Lift

- **Increasing software (s/w) development costs:**
  - Commercial aircraft s/w development cost is \$10B
  - >70% of new aircraft development cost is s/w
  - >70% of s/w development cost in rework and certification
  - S/W complexity increasing logarithmically
- **Obsolescence driven by:**
  - Rapid advancements in computing technology
  - Proliferation of sophisticated threat systems
- **Increasing certification challenges:**
  - Multi-core processors
  - Multi-level Security
  - Integrated Modular Avionics
  - Increasing complexity of Cyber Physical Systems
- **Time to integrate and field new capabilities**
- **Emphasis on commonality across the fleet**
- **Re-use and portability of s/w between on-board and off-board systems**
- **Adequacy/maturity of architecturally centric model based system engineering tools and processes to address challenges**

TECHNOLOGY DRIVEN. WARFIGHTER FOCUSED.

# Objectives of this tutorial

---

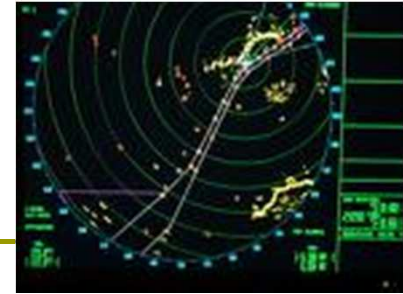


## □ Issues

- How to model/design a real-time critical embedded system that conforms to requirements?
  - How to verify the solution?
  - How to simulate it?
  - How to implement it (not in this tutorial!)?
- One solution among others: use an architecture description language
- to model the system,
  - to run various verification,
  - and to automatically produce the system
- Focus on the AADL2.2 SAE standard

# Objectives of this tutorial

---



- ❑ Illustration: model of a simple radar system
- ❑ Let us suppose we have the following requirements
  1. System implementation **is composed by physical devices** (Hardware entity): antenna + processor + memory + bus
  2. and **software entities** : **running processes and threads** + operating system functionalities (scheduling) implemented in the processor that represent a part of execution platform and physical devices in the same time.
  3. The **main process is responsible for signals processing** : general pattern: **transmitter -> antenna -> receiver -> analyzer -> display**
  4. **Analyzer is a periodic thread** that compares transmitted and received signals to perform detection, localization and identification.
  5. [..]

# Resources for this tutorial

---

## □ Information on AADL

- <http://www.aadl.info> : updates on AADL standard
- <http://www.openaadl.org> : many AADL resources
- <http://www.ellidiss.fr/>: AADLInspector and Ellidiss Tech. AADL activities
- <http://beru.univ-brest.fr/~singhoff/cheddar/>: Cheddar and real-time scheduling

## □ Feel free to contact us for more details

# Outline

---

**Goal:** overview of scheduling analysis capabilities that are proposed by the AADL and tools implementing it. Show the benefits that can be expected by performing early scheduling analysis for real-time software.

- ❑ **Part 1: introduction to AADLv2 core (about 2h/2h30)**
  - Syntax, semantics of the language
- ❑ **Part 2: introducing a case study (about 15')**
  - A radar illustrative case study
- ❑ **Part 3: scheduling analysis (about 2h/2h30)**
  - Introducing real-time scheduling and its use with AADL
- ❑ **Part 4: practical labs, exercises, discussion (about 1 or 2 hours)**
  - How to use tools in order to apply what we learnt in parts 1 to 3

# Presentation of the AADL: Architecture Analysis and Design Language



# Outline

---

1. **AADL a quick overview**
2. AADL key modeling constructs
  1. AADL components
  2. Properties
  3. Component connection
  4. Behavior annex
3. AADL: tool support

# Introduction

---

- **ADL, Architecture Description Language:**
  - **Goal** : modeling software and hardware architectures to master complexity ... to perform analysis
  - **Concepts** : components, connections, deployments.
  - **Many ADLs** : formal/non formal, application domain, ...
  
- **ADL for real-time critical embedded systems: AADL**  
(*Architecture Analysis and Design Language*).



# AADL: Architecture Analysis & Design Language

---

- ❑ International standard promoted by SAE, AS-2C committee, released as AS5506 family of standards
- ❑ Core language document:
  - AADL 1.0 (AS 5506) 2004
  - AADL 2.0 (AS 5506A) 2009
  - AADL 2.1 (AS 5506B) 2012
  - AADL 2.2 (AS 5506C) 2017
- ❑ Annex documents to address specific concerns
  - Annex A: ARINC 653 Interface (AS 5506/1A) 2015
  - Annex B: Data Modelling (AS 5506/2) 2011
  - Annex C: Code Generation Annex (AS 5506/1A) 2015
  - Annex D: Behavior Annex v2 (AS 5506/3) 2017
  - Annex E: Error Model Annex v2 (AS 5506/1A) 2015



## AADL is for Analysis

---

- **AADL objectives are “to model a system”**
  - With analysis in mind (different analysis)
  - To ease transition from well-defined requirements to the final system : code production
  
- Require semantics => any AADL entity has semantics (natural language or formal methods).

# AADL: Architecture Analysis & Design Language

---

- Different representations :
  - **Textual (standardized representation),**
  - Graphical (declarative and instance views),
  - XML/XMI (not part of the standard: tool specific)
  
- Graphical editors:
  - OSATE (SEI):
    - declarative model editor
    - instance model viewer
  - MASIW (ISPRAS)
  - Scade Architect (Ansys): instance model editor
  - Stood for AADL (Ellidiss) : instance model editor

# AADL components

---

- **AADL model** : hierarchy/tree of components
  - Composition hierarchy (subcomponents)
  - Inheritance hierarchy (extends)
  - Binding hierarchy (e.g. process->processor)
  
- **AADL component:**
  - Model a software or a hardware entity
  - May be organized in packages : **reusable**
  - Has a type/interface, zero, one or several implementations
  - May have subcomponents
  - May combine/extend/refine others
  - May have properties : valued typed attributes (source code file name, priority, execution time, memory consumption, ...)
  
- **Component interactions :**
  - Modeled by component connections
  - Binding properties express allocation of SW onto HW

# AADL components

---

## □ **How to declare a component:**

- Component type: name, category, properties, features => interface
- Component implementation: internal structure (subcomponents), properties

## □ **Component categories:** model real-time abstractions, close to the implementation space (ex : processor, task, ...). Each category has well-defined semantics/behavior, refined through the property and annexes mechanisms

- Hardware components: execution platform
- Software components
- Systems : bounding box of a system. Model deployments.

# Component type

---

- Specification of a component: interface
- All component type declarations follow the same pattern:

`<category> foo [extends <bar>]` ←

Inherit features and properties from parent

## **features**

`-- list of features` ←

`-- interface`

Interface of the component:  
Exchange messages, access to  
data or call subprograms

## **properties**

`-- list of properties`

`-- e.g. priority` ←

Some properties describing  
non-functional aspect of the  
component

`end foo;`

# Component type

---

## □ Example:

```
subprogram Spg
features
  in_param : in parameter foo_data;
properties
  Source_Language => C;
  Source_Text => ("foo.c");
end Spg;
```

-- model a sequential execution flow  
-- *Spg* represents a C function,  
-- in file "foo.c", that takes one  
-- parameter as input

← Standard properties, one can define its own properties

```
thread bar_thread
features
  in_data : in event data port foo_data;
properties
  Dispatch_Protocol => Sporadic;
end bar_thread;
```

-- model a schedulable flow of control  
-- *bar\_thread* is a sporadic thread :  
-- dispatched whenever it  
-- receives an event on its "in\_data"  
-- port

# Component implementation

---

- Implementation of a component: body
  - Think spec/body package (Ada), interface/class (Java)

<category> **implementation** foo.i [**extends** <bar>.i]

**subcomponents**

...

**calls**

-- *subprogram subcomponents*

-- *called, only for threads or subprograms*

**connections**

**properties**

-- *list of properties, e.g. Deadline*

**end** foo.i;

foo.i implements foo





# Component implementation

---

## □ Example:

```
subprogram Spg
features
  in_param : in parameter foo_data;
properties
  Source_Language => C;
  Source_Text => ("foo.c");
end Spg;
```

```
thread bar_thread
features
  in_data : in event data port foo_data;
properties
  Dispatch_Protocol => Sporadic;
end bar_thread;
```

Connect  
data/parameter

**thread implementation** bar\_thread.impl  
**calls**

```
C : { S : subprogram spg; };
connections
  parameter in_data -> S.in_param;
end bar_thread.impl;
```

*-- in this implementation, at each  
-- dispatch we execute the "C" call  
-- sequence. We pass the dispatch  
-- parameter to the call sequence*

# AADL concepts

---

- **AADL introduces many other concepts:**
  - Related to embedded real-time critical systems :
    - AADL flows: capture high-level data+control flows
    - AADL modes: model operational modes in the form of an alternative set of active components/connections/...
  - To ease models design/management:
    - AADL packages (similar to Ada/Java, renames, private/public)
    - AADL abstract component, component extension
    - ...
- **AADL is a rich language :**
  - Around 200 entities in the meta-model
  - Around 200 syntax rules in the BNF (core)
  - Around 250 legality rules and more than 500 semantics rules
  - 355 pages core document + various annex documents

# Outline

---

1. AADL a quick overview
2. **AADL key modeling constructs**
  1. **AADL components**
  2. Properties
  3. Component connection
  4. Behavior annex
3. AADL: tool support

# AADL workflow

---

## 1. Declarative model (Packages)

- HW libraries
- SW libraries
- Applicative composite systems

bottom-up

top-down

similar to  
UML classes  
or SysML blocks

## 2. Instance model

- Selection of the Root System
- Expanded HW hierarchy
- Expanded SW hierarchy

exhaustive  
representation of  
the system  
hierarchy

## 3. Deployed model

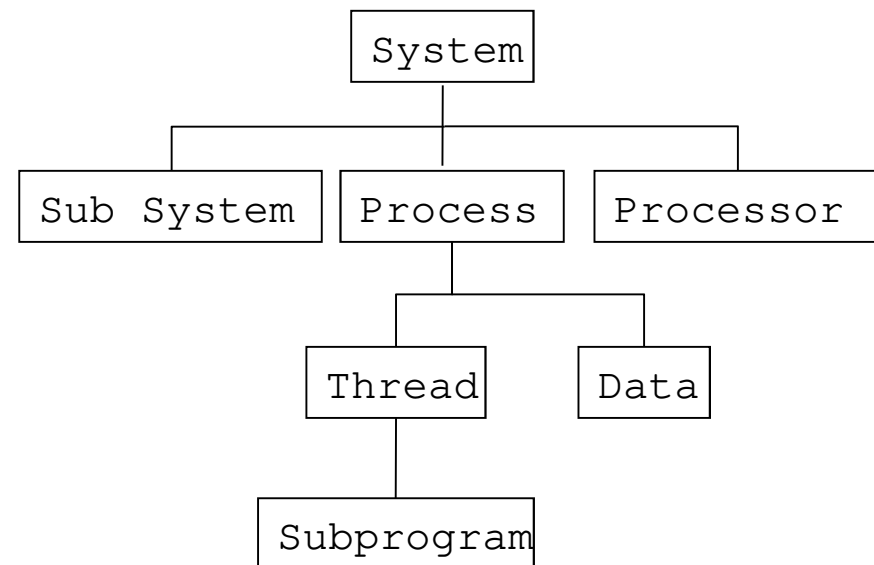
- SW instances binding onto HW instances

required for many  
advanced analysis:  
-schedulability  
-simulation  
-safety  
-security  
-...

# A full AADL system : a tree of component instances

---

- ❑ Component types and implementations only define a library of entities (classifiers)
- ❑ An AADL model is a set of component instances (of the classifiers)
- ❑ System must be instantiated through a hierarchy of subcomponents, from root (system) to the leafs (subprograms, ..)
- ❑ We must choose a system implementation component as the root system model !



# Software components categories

---

- ❑ **thread** : schedulable execution flow, Ada or VxWorks task, Java or POSIX thread. Execute programs
- ❑ **data** : data placeholder, e.g. C struct, C++ class, Ada record
- ❑ **process** : address space. It must hold at least one thread
- ❑ **subprogram** : a sequential execution flow. Associated to a source code (C, Ada) or a model (SCADE, Simulink)
- ❑ **thread group** : hierarchy of threads
- ❑ **subprogram group** : library or hierarchy of subprograms



# Software components

---

- **Example of a process component** : composed of two threads

```
thread receiver  
end receiver;
```

```
thread implementation receiver.impl  
end receiver.impl;
```

```
thread analyser  
end analyser;
```

```
thread implementation analyser.impl  
end analyser.impl;
```

```
process processing  
end processing;
```

```
process implementation processing.others  
subcomponents
```

```
  receive : thread receiver.impl;  
  analyse : thread analyser.impl;
```

```
  . . .
```

```
end processing.others;
```

# Software components

---

- **Example of a thread component** : a thread may call different subprograms

```
subprogram Receiver_Spg  
end Receiver_Spg;
```

```
subprogram ComputeCRC_Spg  
end ComputeCRC_Spg;
```

...

```
thread receiver  
end receiver;
```

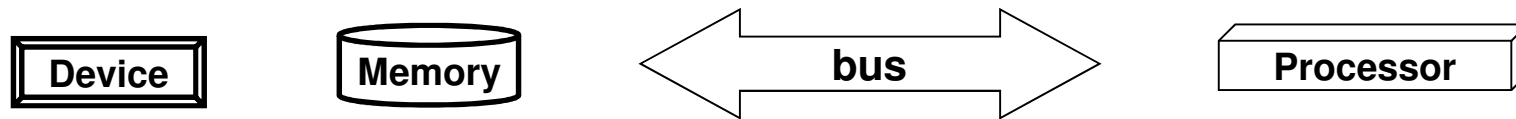
```
thread implementation receiver.impl  
CS : calls {  
    call1 : subprogram Receiver_Spg;  
    call2 : subprogram ComputeCRC_Spg;  
};  
end receiver.impl;
```



# Hardware components categories

---

- ❑ **processor/virtual processor** : scheduling component (combined CPU and OS scheduler).
- ❑ **memory** : model data storage (memory, hard drive)
- ❑ **device** : component that interacts with the environment. Internals (e.g. firmware) is not modeled.
- ❑ **bus/virtual bus** : data exchange mechanism between components



## « system » category

---

### □ ***system*** :

1. Help structuring an architecture, with its own hierarchy of subcomponents. A system can include one or several subsystems.
2. Root system component.
3. Bindings : model the deployment of components inside the component hierarchy.

System

## « system » category

---

```
subprogram Receiver_Spg ...  
thread receiver ...  
  
thread implementation receiver.impl  
  call1 : subprogram Receiver_Spg;  
  ...  
end receiver.impl;  
  
process processing  
end processing;  
  
process implementation processing.others  
subcomponents  
  receive : thread receiver.impl;  
  analyse : thread analyser.impl;  
  ...  
end processing.others;
```

```
device antenna  
end antenna;
```

```
processor leon2  
end leon2;
```

---

```
system radar  
end radar;
```

```
system implementation radar.simple  
subcomponents
```

```
  main : process processing.others;  
  cpu : processor leon2;
```

```
properties
```

```
  Actual_Processor_Binding =>  
    reference cpu applies to main;  
end radar.simple;
```

# About subcomponents

---

- Semantics: restrictions apply on subcomponents
  - e.g. hardware cannot contain software, etc

<b>category</b>	<b>allowed subcomponent categories</b>
system	all but thread group and thread
processor	virtual processor, memory, bus
memory	memory, bus
process	thread group, thread, subprogram, data
thread group	thread group, thread, subprogram, data
thread	subprogram, data
subprogram	data
data	data, subprogram

# Outline

---

1. AADL a quick overview
2. **AADL key modeling constructs**
  1. AADL components
  2. **Properties**
  3. Component connection
  4. Behavior annex
3. AADL: tool support

# AADL properties

---

## □ **Property:**

- Typed attribute, associated to one or more entities
  - Property definition = name + type + possible owners
  - Property association to a component = property name + value
- Can be propagated to subcomponents: **inherit**
  - Can override parent's one, case of extends

## □ **Allowed types in properties:**

- **aadlboolean, aadlinteger, aadlreal, aadlstring, range, list, enumeration, record**, user defined (Property type)

# AADL properties

---

## □ Property sets :

- Group property definitions.
- Property sets part of the standard, e.g. Thread\_Properties.
- Or user-defined, e.g. for new analysis as power analysis

## □ Example :

**property set** Thread\_Properties **is**

...

Priority : **aadlinteger** **applies to** (thread, device, ...);

Source\_Text : **inherit list of aadlstring** **applies to** (data, port, thread, ...);

...

**end** Thread\_Properties;

# AADL properties

---

- Properties are typed with units to model physical systems, related to embedded real-time critical systems.

```
property set AADL_Projects is
```

```
Time_Units: type units (
```

```
  ps,
```

```
  ns => ps * 1000,
```

```
  us => ns * 1000,
```

```
  ms => us * 1000,
```

```
  sec => ms * 1000,
```

```
  min => sec * 60,
```

```
  hr => min * 60);
```

```
--
```

```
end AADL_Projects;
```

```
property set Timing_Properties is
```

```
Time: type aadlinteger
```

```
  0 ps .. Max_Time units Time_Units;
```

```
Time_Range: type range of Time;
```

```
Compute_Execution_Time: Time_Range
```

```
applies to (thread, device, subprogram,  
  event port, event data port);
```

```
end Timing_Properties;
```



# AADL properties

---

- Properties can apply to (*with increasing priority*)
  - a component type (1)
  - a component implementation (2)
  - a subcomponent (3)
  - a contained element path (4)

```
thread receiver
properties -- (1)
  Compute_Execution_Time => 3 ms .. 4 ms;
  Deadline => 150 ms ;
end receiver;
```

```
thread implementation receiver.impl
properties -- (2)
  Deadline => 160 ms;
end receiver.impl;
```

```
process implementation processing.others
subcomponents
  receive0 : thread receiver.impl;
  receive1 : thread receiver.impl;
  receive2 : thread receiver.impl
    {Deadline => 200 ms;}; -- (3)
properties -- (4)
  Deadline => 300 ms applies to receive1;
end processing.others;
```

# Outline

---

1. AADL a quick overview
2. **AADL key modeling constructs**
  1. AADL components
  2. Properties
  3. **Component connection**
  4. Behavior annex
3. AADL: tool support

# Component connection

---

- ❑ **Connection:** model component interactions, control flow and/or data flow. E.g. exchange of messages, access to shared data, remote subprogram call (RPC), ...
- ❑ **features :** connection point part of the interface. Each *feature* has a name, a direction, and a category
- ❑ **Features category:** specification of the type of interaction
  - *event port:* event exchange (e.g. alarm, interrupt)
  - *data port:* data exchange triggered by the scheduler
  - *event data port:* data exchange of data triggered with sender (message)
  - *subprogram parameter*
  - *data access :* access to external data component, possibly shared
  - *subprogram access :* RPC or rendez-vous
- ❑ **Features direction for port and parameter:**
  - input (*in*), output (*out*), both (*in out*).

# Component connection

---

- ❑ Features of subcomponents are connected in the “connections” subclause of the enclosing component
- ❑ Ex: threads & thread connection on data port

```
thread analyser
```

```
features
```

```
  analyser_out : out data port  
    Target_Position.Impl;
```

```
end analyser;
```

```
thread display_panel
```

```
features
```

```
  display_in : in data port Target_Position.Impl;  
end display_panel;
```

```
process implementation processing.others
```

```
subcomponents
```

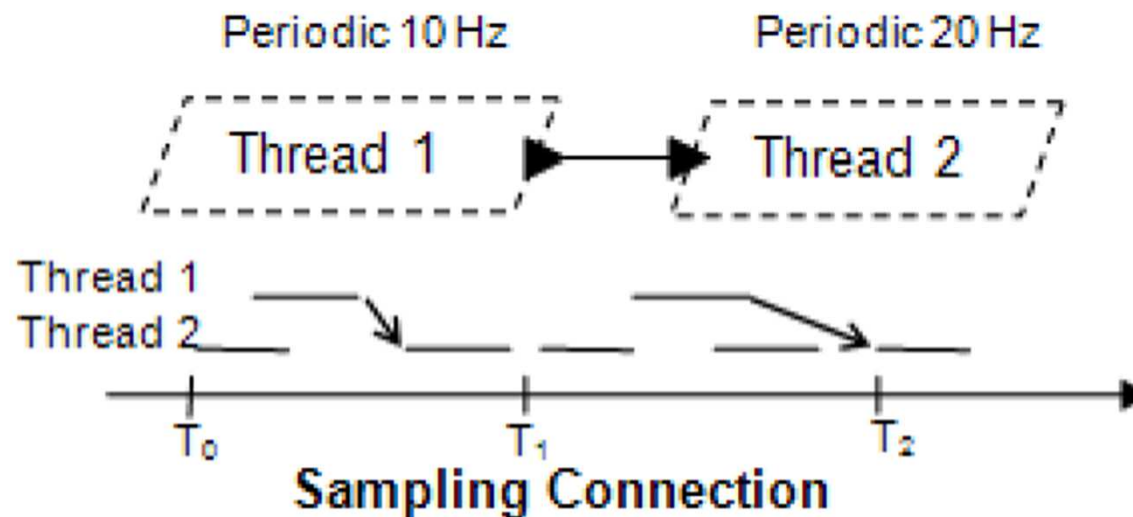
```
  display : thread display_panel.impl;  
  analyse : thread analyser.impl;
```

```
connections
```

```
  port analyse.analyser_out -> display.display_in;  
end processing.others;
```

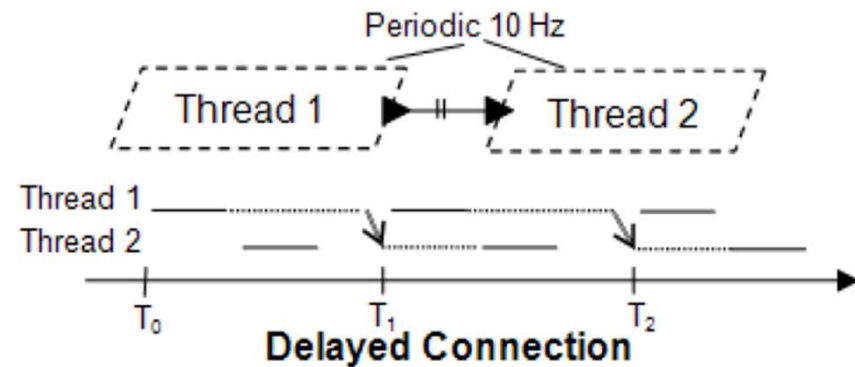
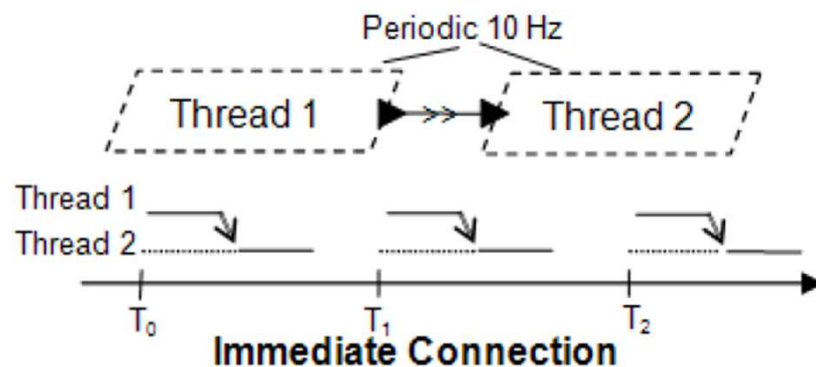
# Data connection policies

- ❑ Allow deterministic communications
- ❑ Multiple policies exist to control production and consumption of data by threads:
  1. **Sampling connection:** takes the latest value
    - ❑ Problem: data consistency (lost or read twice) !



# Data connection policies

- 2. **Immediate:** receiver thread is immediately awoken, and will read data when emitter finished
- 3. **Delayed:** actual transmission is delayed to the next time frame



# Component connection

---

## □ Connection for shared data :

```
process implementation processing.others
```

```
subcomponents
```

```
  analyse : thread analyser.impl;
```

```
  display : thread display_panel.impl;
```

```
  a_data : data shared_var.impl;
```

```
connections
```

```
  cx1 : data access a_data -> display.share;
```

```
  cx2 : data access a_data -> analyse.share;
```

```
end processing.others;
```

```
data shared_var
```

```
end shared_var;
```

```
data implementation shared_var.impl
```

```
end shared_var.impl;
```

```
thread analyser
```

```
features
```

```
  share : requires data access shared_var.impl;
```

```
end analyser;
```

```
thread display_panel
```

```
features
```

```
  share : requires data access shared_var.impl;
```

```
end display_panel;
```

# Component connection

---

## □ Connection for shared data :

```
process implementation processing.others
```

```
subcomponents
```

```
  analyse : thread analyser.impl;
```

```
  display : thread display_panel.impl;
```

```
  a_data : data shared_var.impl;
```

```
connections
```

```
  cx1 : data access a_data -> display.share;
```

```
  cx2 : data access a_data -> analyse.share;
```

```
end processing.others;
```

```
data shared_var
```

```
end shared_var;
```

```
data implementation shared_var.impl
```

```
end shared_var.impl;
```

```
thread analyser
```

```
features
```

```
  share : requires data access shared_var.impl;
```

```
end analyser;
```

```
thread display_panel
```

```
features
```

```
  share : requires data access shared_var.impl;
```

```
end display_panel;
```



# Component connection

---

## □ Connection between *thread* and *subprogram* :

```
thread implementation receiver.impl  
calls {  
  RS: subprogram Receiver_Spg;  
};  
connections  
  parameter RS.receiver_out -> receiver_out;  
  parameter receiver_in -> RS.receiver_in;  
end receiver.impl;
```

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

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

# Outline

---

1. AADL a quick overview
2. **AADL key modeling constructs**
  1. AADL components
  2. Properties
  3. Component connection
  4. **Behavior annex**
3. AADL: tool support

## AADL Behavior Annex

---

- ❑ Provides more details on the internal behavior of threads and subprograms.
- ❑ Complements, extends or replaces Modes, Calls and some Properties defined in the core model.
- ❑ Required for accurate timing analysis and virtual execution of the AADL model.
- ❑ State Transition Automata with an action language:
  - dispatch conditions
  - actions: event sending, subprogram call, critical sections, ...
  - control structures: loops, tests, ...

# AADL Behavior Annex example

---

```
thread transmitter  
features  
  transmitter_out : out data port radar_types::Radar_Pulse;  
end transmitter;
```

```
thread implementation transmitter.impl
```

```
...
```

```
annex Behavior_Specification {**
```

```
  states
```

```
    s : initial complete final state;
```

```
  transitions
```

```
    t : s -[on dispatch]-> s { transmitter_out := "ping" };
```

```
**};
```

```
end transmitter.impl;
```

annex identifier

state declaration

transition condition

transition actions

# Outline

---

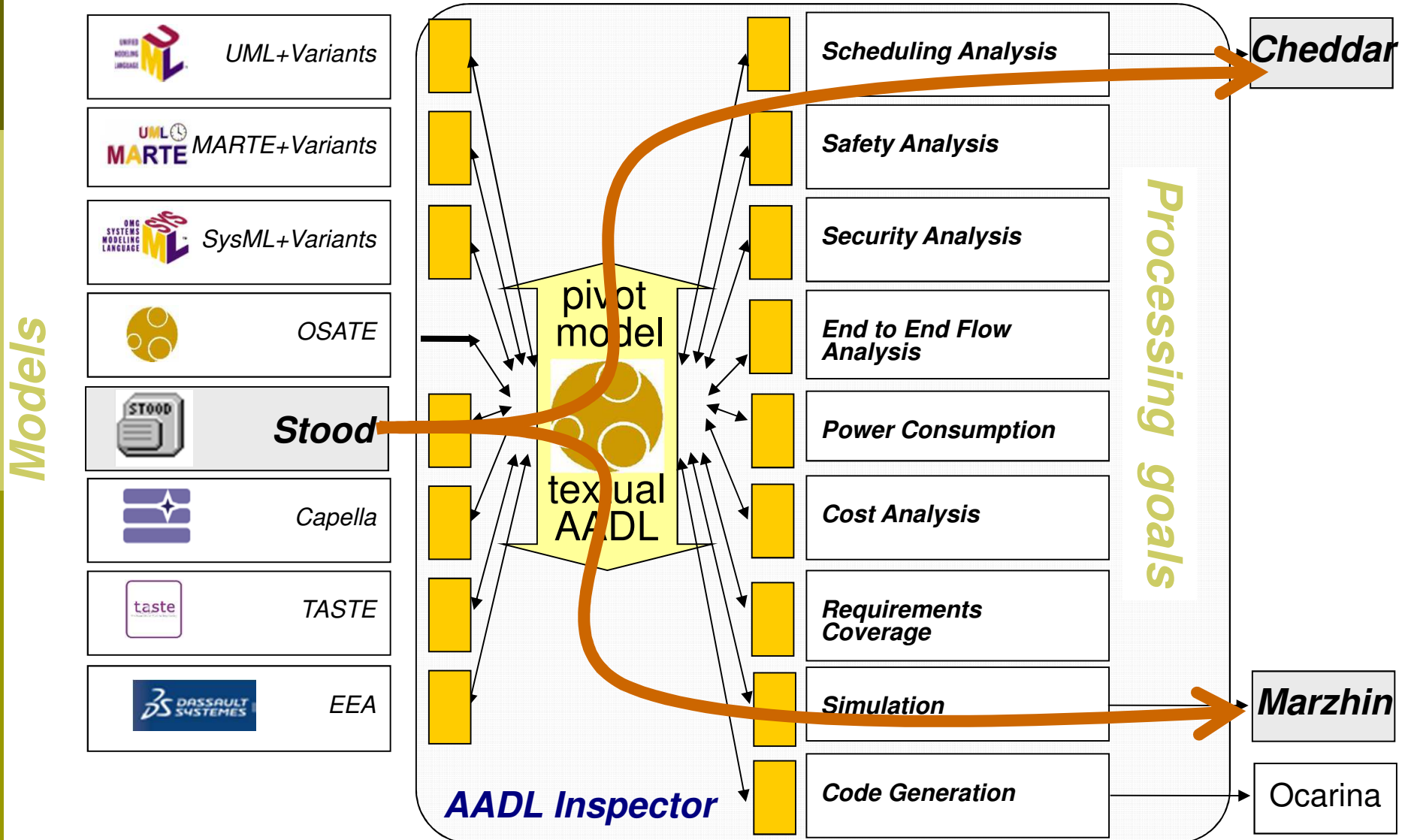
1. AADL a quick overview
2. AADL key modeling constructs
  1. AADL components
  2. Properties
  3. Component connection
  4. Behavior annex
3. **AADL: tool support**

# AADL & Tools

---

- **OSATE** (SEI/CMU, <http://aadl.info>)
  - Eclipse-based tools. Reference implementation.
  - Textual and graphical editors + various analysis plug-ins
- **STOOD** (Ellidiss, <http://www.ellidiss.com> )
  - Graphical editor, code/documentation generation
  - Guided modeling approach, requirements traceability
- **Cheddar** (UBO/Lab-STICC, <http://beru.univ-brest.fr/~singhoff/cheddar/> )
  - Performance analysis
- **AADLInspector** (Ellidiss, <http://www.ellidiss.com>)
  - Standalone framework to process AADL models and Behavior Annex
  - Industrial version of Cheddar + Simulation Engine
- **Ocarina** (ISAE, <http://www.openaadl.org>)
  - Command line tool, library to manipulate models.
  - AADL parser + code generation + analysis (Petri Net, WCET, ...)
- **Others:** RAMSES, PolyChrony, ASSIST, MASIW, MDCF, TASTE, Scade Architect, Camet, Bless, ...

# Tools used for the tutorial



# Tools used for the tutorial

## □ AADLInspector, OSATE/Cheddar

test	entity	
<input checked="" type="checkbox"/> task response time computed from simulation	cpu	No deadline mis
Number of preemptions	cpu	4
Number of context switches	cpu	74
Task response time computed from simulation	cpu.partition1_pr.T	worst = 5, best =
Task response time computed from simulation	cpu.partition1_pr.T	worst = 15, best =
Task response time computed from simulation	cpu.partition2_pr.T	worst = 15, best =
<input type="checkbox"/> Set priorities according to Rate Monotonic	cpu	
<input type="checkbox"/> Set priorities according to Deadline Monotoni	cpu	

**Scheduling simulation, Processor arinc :**

- Number of preemptions : 760
- Number of context switches : 3205
- Task response time computed from simulation :
  - T1 => 6/worst 6/best 6.00000/average
  - T2 => 56/worst 35/best 46.81667/average
  - T3 => 10/worst 4/best 6.00000/average
  - T4 => 1/worst 1/best 1.00000/average
- No deadline missed in the computed scheduling : the task set seems to be schedulable.

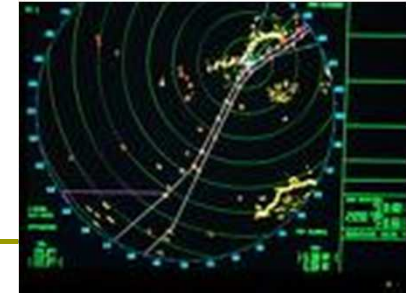


# AADL: a radar case study



## Back to radar case study

---



- Goal: to model a simple radar system
- Let us suppose we have the following requirements
  1. System implementation **is composed by physical devices** (Hardware entity): antenna + processor + memory + bus
  2. and **software entity : running processes and threads** + operating system functionalities (scheduling) implemented in the processor that represent a part of execution platform and physical devices in the same time.
  3. The **main process is responsible for signals processing** : general pattern: **transmitter -> antenna -> receiver -> analyzer -> display**
  4. **Analyzer is a periodic thread** that compares transmitted and received signals to perform detection, localization and identification.
  5. [..]

# Tools used for modeling

---

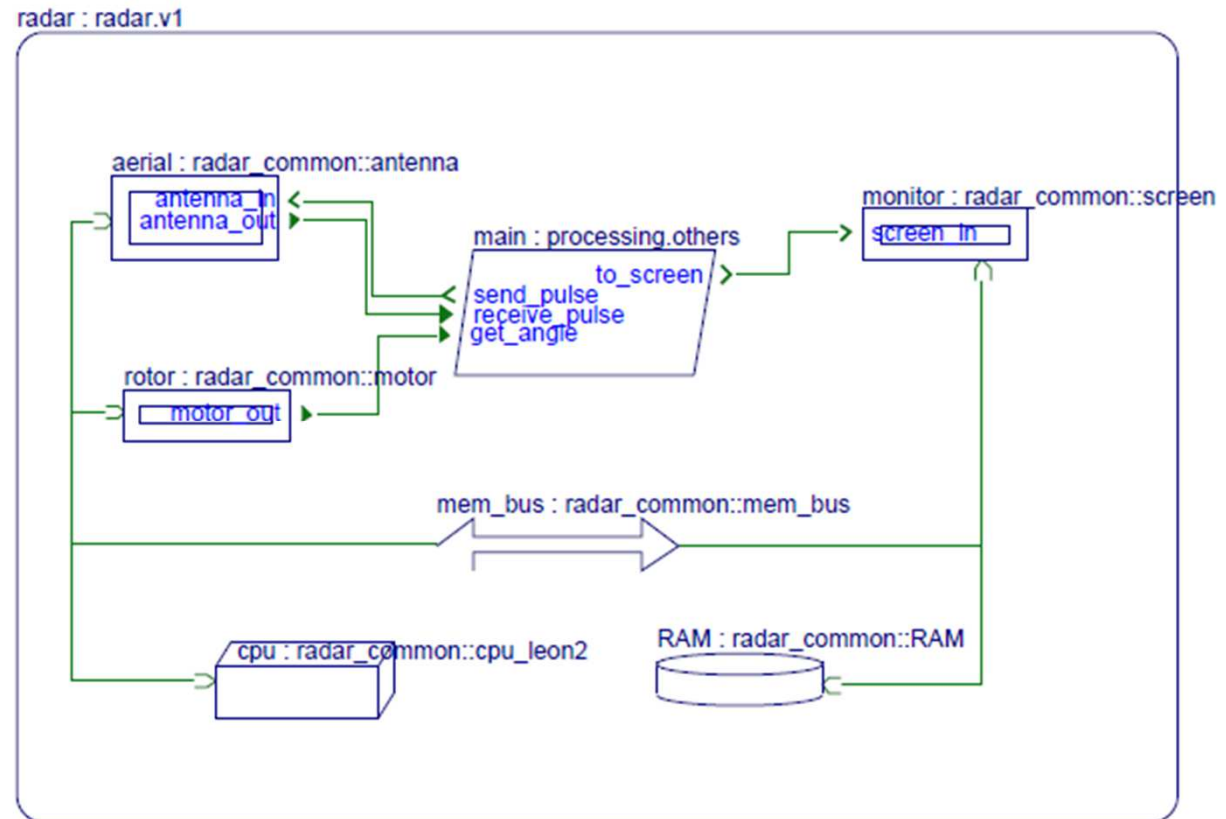
- ❑ AADL syntax is both textual and graphical, with several editors available
  - Modes exist for emacs, vi
  - OSATE provides a comprehensive textual IDE on top of Eclipse, and additional plug-ins
    - ❑ IMV : Instance Model Viewer
    - ❑ Consistency checkers, statistics, various analysis.
  - Stood for AADL:
    - ❑ Top-down modeling approach
    - ❑ Instance Model graphical editor
    - ❑ Generation of textual AADL for tool interoperability
  - ...
- ❑ In the following, we will use Stood

# Radar case study

## Hardware/Software breakdown: components

```
PACKAGE radar_v1
PUBLIC
-- ...
SYSTEM radar
END radar;
-- ...
PROCESS processing
-- ...
END processing;
-- ...
END radar_v1;

PACKAGE radar_common
PUBLIC
-- ...
DEVICE screen
-- ...
END screen;
-- ...
END radar_common;
```



# Radar case study

## Hardware/Software breakdown: features

```
PROCESS processing
```

```
FEATURES
```

```
to_screen : OUT EVENT PORT;  
send_pulse : OUT EVENT PORT;  
receive_pulse : IN DATA PORT;  
get_angle : IN DATA PORT;
```

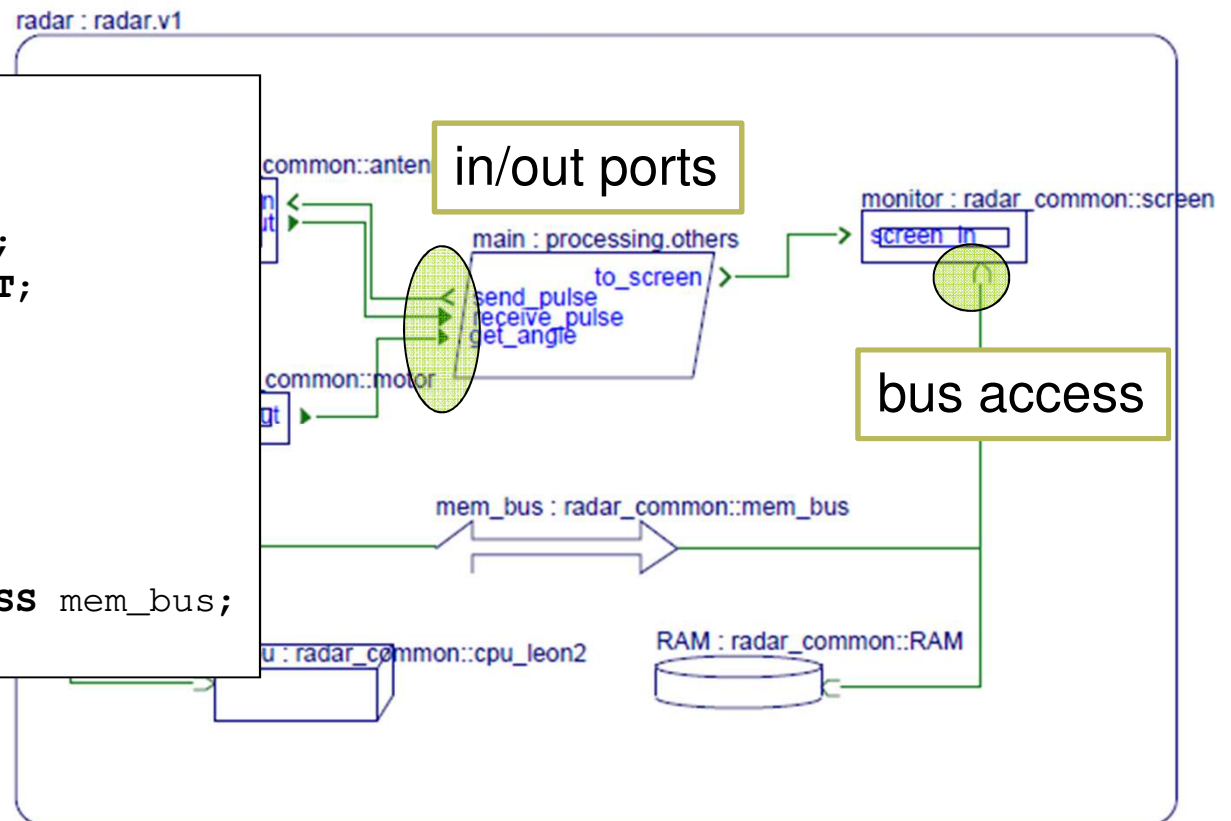
```
END processing;
```

```
DEVICE screen
```

```
FEATURES
```

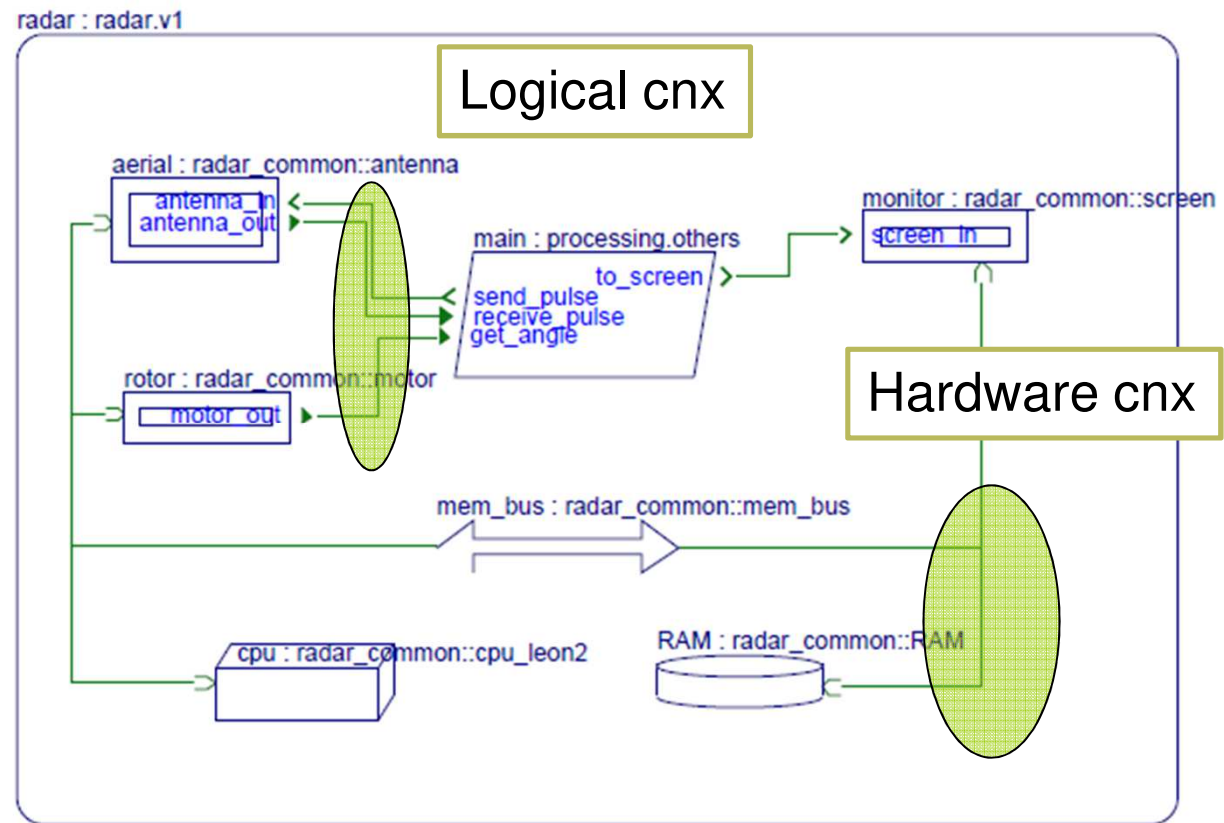
```
screen_in : IN EVENT PORT;  
mem_bus : REQUIRES BUS ACCESS mem_bus;
```

```
END screen;
```



# Radar case study

## Hardware/Software breakdown: connections



*note:*  
*bindings are not represented graphically with Stood*

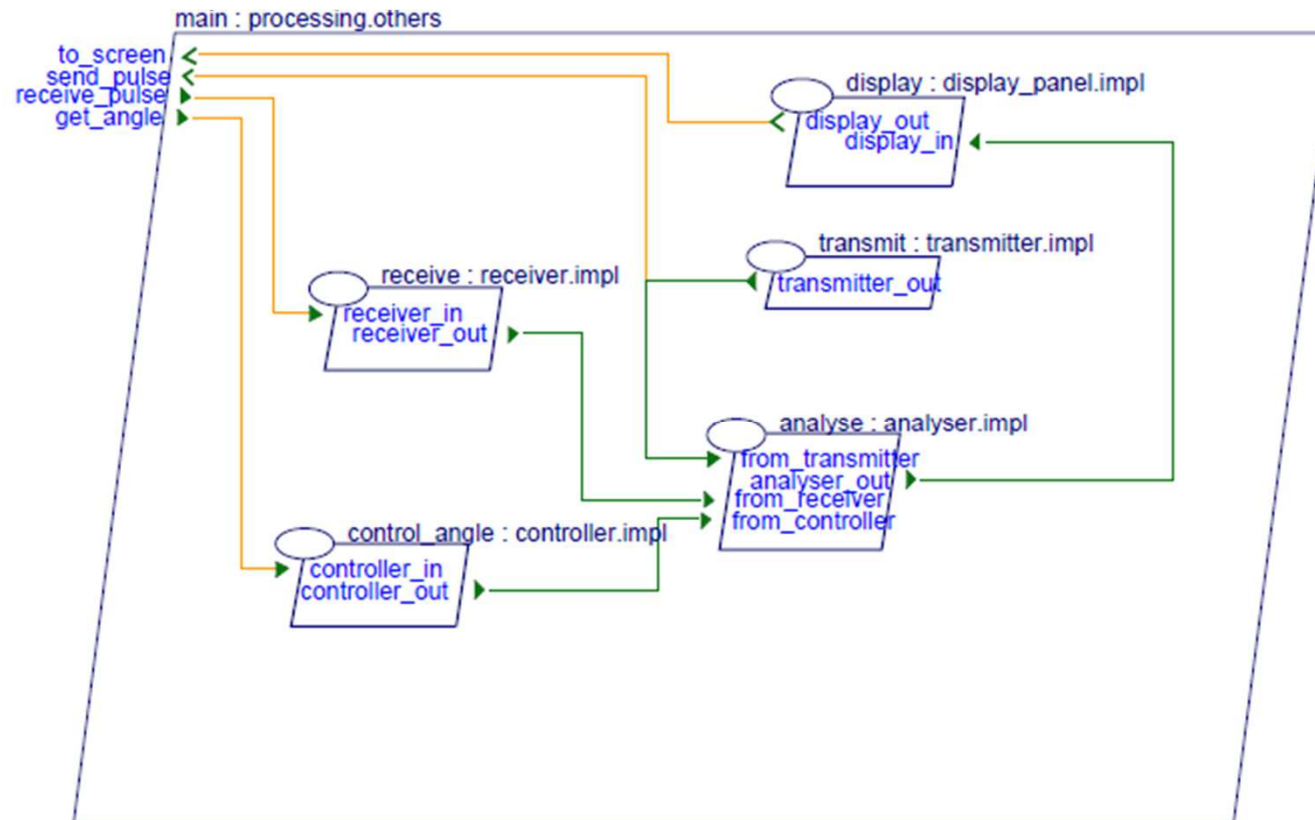
# Radar case study

## □ Hardware/Software breakdown: connections

```
SYSTEM IMPLEMENTATION radar.v1
SUBCOMPONENTS
  aerial : DEVICE radar_common::antenna;
  rotor  : DEVICE radar_common::motor;
  monitor : DEVICE radar_common::screen;
  cpu    : PROCESSOR radar_common::cpu_leon2;
  mem_bus : BUS radar_common::mem_bus;
  RAM    : MEMORY radar_common::RAM;
  main   : PROCESS processing.others;
CONNECTIONS
  cnx1 : PORT aerial.antenna_out -> main.receive_pulse;
  cnx2 : PORT rotor.motor_out   -> main.get_angle;
  cnx3 : PORT main.send_pulse   -> aerial.antenna_in;
  cnx4 : PORT main.to_screen    -> monitor.screen_in;
  cnx5 : BUS ACCESS mem_bus     -> aerial.mem_bus;
  cnx6 : BUS ACCESS mem_bus     -> rotor.mem_bus;
  cnx7 : BUS ACCESS mem_bus     -> monitor.mem_bus;
  cnx8 : BUS ACCESS mem_bus     -> cpu.mem_bus;
  cnx9 : BUS ACCESS mem_bus     -> RAM.mem_bus;
  -- ...
END radar.v1;
```

# Radar case study

## Software elements





## A few words on AADL usage

---

- AADL is for architectural description and early analysis
  
- Not to be compared with UML suites
  - Not a graphical representation of the source code
  - But can be associated with existing source code via Properties
  
- Keep in mind models support an objective
  - For now, it is just a high-level view of the design
  
- In the next sections, we will complete the models with properties to support schedulability analysis

# AADL: about scheduling analysis



# Real-time Scheduling analysis/theory, what is it?

---

- ❑ **Embedded real-time critical systems** have temporal constraints to meet (e.g. deadline).
- ❑ Many systems are built with operating systems providing multitasking facilities ... Tasks may have deadline.
- ❑ **But, tasks make temporal constraints analysis difficult to do:**
  - ❑ We must take interference delaying a task into account: other tasks, shared resources, ...
  - ❑ Need to take scheduling into account.
  - ❑ Scheduling (or schedulability) analysis.

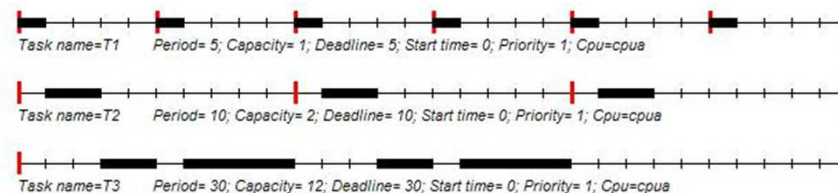
# Real-Time scheduling theory

1. **A set of simplified tasks models** (to model functions of the system)
2. **A set of analytical methods** (called feasibility tests)

- **Example:**

$$R_i \leq \text{Deadline} \quad R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{P_j} \right\rceil \cdot C_j$$

3. **A set of scheduling algorithms:** build the full scheduling/GANTT diagram



# Real-Time scheduling theory is hard to apply

---

- Real-Time scheduling theory (uniprocessor)
  - Theoretical results defined from 1974 to 1994:  
feasibility tests exist for uniprocessor architectures
- Supported at a decent level since POSIX 1003 real-time operating systems and ARINC653, ...
- Industry demanding
  - Yet, hard to use

# Summary

---

1. Issues about real-time scheduling analysis: AADL to the rescue
2. Basics on scheduling analysis: fixed-priority scheduling for uniprocessor architectures
3. AADL components/properties to scheduling analysis

# What to model to achieve early scheduling analysis

---

## 1. **Software side:**

- Workload: release time, execution time
- Timing constraints
- Software entity interferences, examples:
  - Tasks relationships/communication or synchronization: e.g. shared data, data flow
  - Task containers: ARINC 653 partition, process

## 2. **Hardware (should be called execution platform) side:**

- Available resources, e.g. computing capabilities
- Contention, interference, examples: processing units, cache, memory bus, NoC, ...

## 3. **Deployment**

**=> Architecture models**

**=> It is the role of an ADL to model those elements**

# Real-Time scheduling theory is hard to apply

---

- Requires strong theoretical knowledge/skills
  - Numerous theoretical results: how to choose the right one?
  - Numerous assumptions for each result.
  - How to abstract/model a system to verify deadlines?
- How to integrate scheduling analysis in the engineering process?
  - When to apply it? What about tools?

**=> It is the role of an ADL to hide those details**



# AADL to the rescue?

---

## □ Why AADL helps:

### ■ All required model elements are given for the analysis

- Component categories: thread, data, processor
- Feature categories: data access, data port, ...
- Properties: Deadline, Priority, WCET, Ceiling Priority, ...
- Annexes (e.g. behavior annex)

### ■ AADL semantic: formal and natural language

- E.g. automata to define the concept of periodic thread
- Close to the real-time scheduling analysis methods

### ■ Model engineering: reusability, several levels of abstraction

### ■ Tools & chain tools: AADL as a pivot language (international standard)

- VERSA, OSATE, POLA/FIACRE/TINA, CARTS, MAST, Marzhin, Cheddar, ... by Ocarina, TASTE, AADLInspector, RAMSES, MOSART, OSATE ...

# AADL to the rescue?

---

## □ **But AADL does not solve everything:**

- AADL is a complex language
- How to ensure model elements are compliant with analysis requirements/assumptions, **sustainability**, accuracy, ...
- Not a unique AADL model for a given system to model
- Not a unique mapping between a design model and an analysis model
- Having AADL scheduling analysis tools is not enough too, how to use them?
- ...

# Summary

---

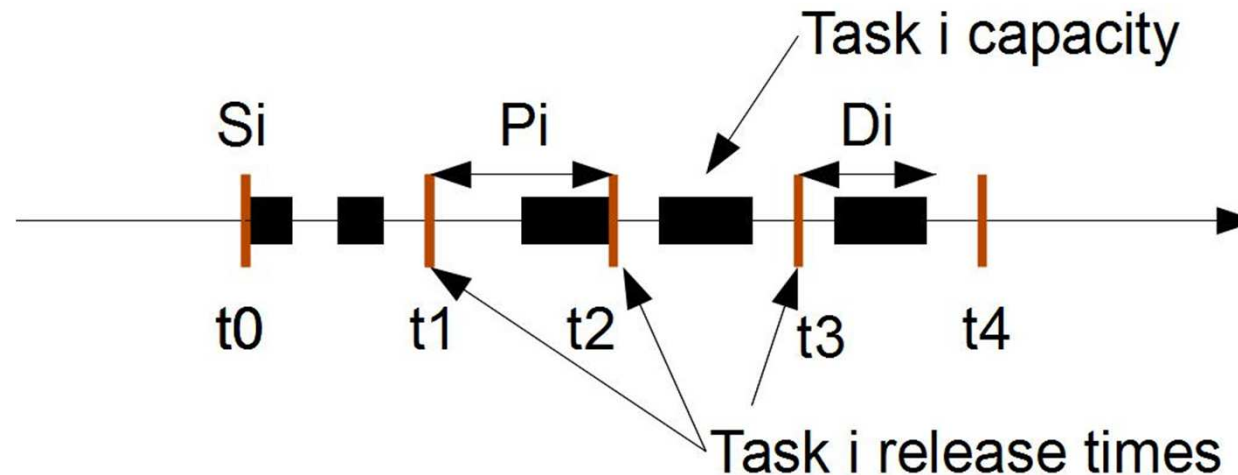
1. Issues about real-time scheduling analysis: AADL to the rescue
2. Basics on scheduling analysis: fixed-priority scheduling for uniprocessor architectures
3. AADL components/properties to scheduling analysis

# Real-time scheduling theory : models of task

---

- **Task simplified model:** sequence of statements + data.
  
- **Usual kind of tasks:**
  - Independent tasks or dependent tasks.
  - Periodic and sporadic tasks (critical functions) : have several jobs and release times
  - Aperiodic tasks (non critical functions) : only one job and one release time

# Real-time scheduling theory : models of task



## □ Usual parameters of a periodic task i:

- **Period:**  $P_i$  (duration between two release times). A task starts a job for each release time.
- **Deadline to meet:**  $D_i$ , timing constraint to meet.
- **First task release time (first job):**  $S_i$ .
- **Worst case execution time of each job:**  $C_i$  (or capacity or WCET).
- **Priority:** allows the scheduler to choose the task to run

## Real-time scheduling theory : models of task

---

- **Assumptions for the next slides (synchronous periodic task with deadlines on requests):**
  - All tasks are periodic.
  - All tasks are independent.
  - $\forall i : P_i = D_i$  : a task must end its current job before its next release time.
  - $\forall i : S_i = 0 \Rightarrow$  called critical instant (worst case on processor demand).

# Uniprocessor fixed priority scheduling

---

## □ **Fixed priority scheduling:**

- Scheduling based on fixed priority => priorities do not change during execution time.
- Priorities are assigned at design time (off-line).
- Efficient and simple feasibility tests.
- Scheduler easy to implement into real-time operating systems.

## □ **Priority assignments:**

- Rate Monotonic, Deadline Monotonic, OPA, ...

# Uniprocessor fixed priority scheduling

---

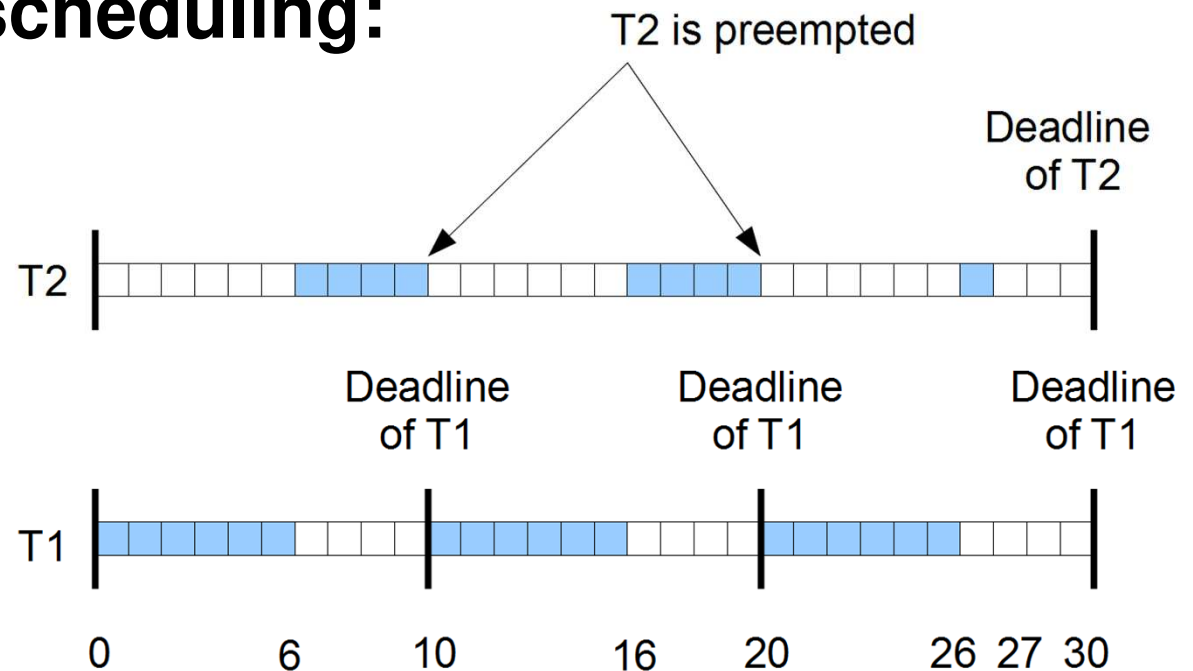
## □ **Rate Monotonic:**

- Optimal priority assignment in the case of fixed priority scheduling and uniprocessor.
- Periodic tasks.
- The highest priority tasks have the smallest periods.



# Uniprocessor fixed priority scheduling

## □ Rate Monotonic assignment and preemptive fixed priority scheduling:



- Assuming VxWorks priority levels (high=0 ; low=255)
- T1 : C1=6, P1=10, Prio1=0
- T2 : C2=9, P2=30, Prio2=1

# Uniprocessor fixed priority scheduling

---

## □ **Feasibility/Schedulability tests to predict on design-time if deadline will be met:**

1. **Run simulations on feasibility interval =  $[0, \text{LCM}(P_i)]$ .**  
Sufficient and necessary condition.

2. **Processor utilization factor test:**

$$U = \sum_{i=1}^n C_i/P_i \leq n \cdot (2^{\frac{1}{n}} - 1) \quad (\text{about } 69\%)$$

Rate Monotonic assignment and preemptive scheduling.  
Sufficient but not necessary condition.

3. **Task worst case response time, noted  $R_i$  :** delay between task release time and task completion time. Any priority assignment, preemptive scheduling.

# Uniprocessor fixed priority scheduling

---

## □ Compute $R_i$ , task $i$ worst case response time:

- Task  $i$  response time = task  $i$  capacity + delay the task  $i$  has to wait for higher priority task  $j$ . Or:

$$R_i = C_i + \sum_{j \in hp(i)} \text{waiting time due to } j \quad \text{or} \quad R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{P_j} \right\rceil \cdot C_j$$

- $hp(i)$  is the set of tasks which have a higher priority than task  $i$ .
- $\lceil x \rceil$  returns the smallest integer not smaller than  $x$ .

## Uniprocessor fixed priority scheduling

---

- To compute task response time: compute  $wi^k$  with:

$$wi^n = Ci + \sum_{j \in hp(i)} \lceil wi^{n-1} / Pj \rceil \cdot Cj$$

- Start with  $wi^0 = Ci$ .
- Compute  $wi^1, wi^2, wi^3, \dots, wi^k$  upto:
  - If  $wi^k > Pi$ . No task response time can be computed for task i. Deadlines will be missed !
  - If  $wi^k = wi^{k-1}$ .  $wi^k$  is the task i response time. Deadlines will be met.

# Uniprocessor fixed priority scheduling

□ **Example:** T1(P1=7, C1=3), T2 (P2=12, C2=2), T3 (P3=20, C3=5)

$$w1^0 = C1 = 3 \Rightarrow R1 = 3$$

$$w2^0 = C2 = 2$$

$$w2^1 = C2 + \left\lceil \frac{w2^0}{P1} \right\rceil \cdot C1 = 2 + \left\lceil \frac{2}{7} \right\rceil \cdot 3 = 5$$

$$w2^2 = C2 + \left\lceil \frac{w2^1}{P1} \right\rceil \cdot C1 = 2 + \left\lceil \frac{5}{7} \right\rceil \cdot 3 = 5 \Rightarrow R2 = 5$$

$$w3^0 = C3 = 5$$

$$w3^1 = C3 + \left\lceil \frac{w3^0}{P1} \right\rceil \cdot C1 + \left\lceil \frac{w3^0}{P2} \right\rceil \cdot C2 = 10$$

$$w3^2 = C3 + \left\lceil \frac{w3^1}{P1} \right\rceil \cdot C1 + \left\lceil \frac{w3^1}{P2} \right\rceil \cdot C2 = 13$$

$$w3^3 = C3 + \left\lceil \frac{w3^2}{P1} \right\rceil \cdot C1 + \left\lceil \frac{w3^2}{P2} \right\rceil \cdot C2 = 15$$

$$w3^4 = C3 + \left\lceil \frac{w3^3}{P1} \right\rceil \cdot C1 + \left\lceil \frac{w3^3}{P2} \right\rceil \cdot C2 = 18$$

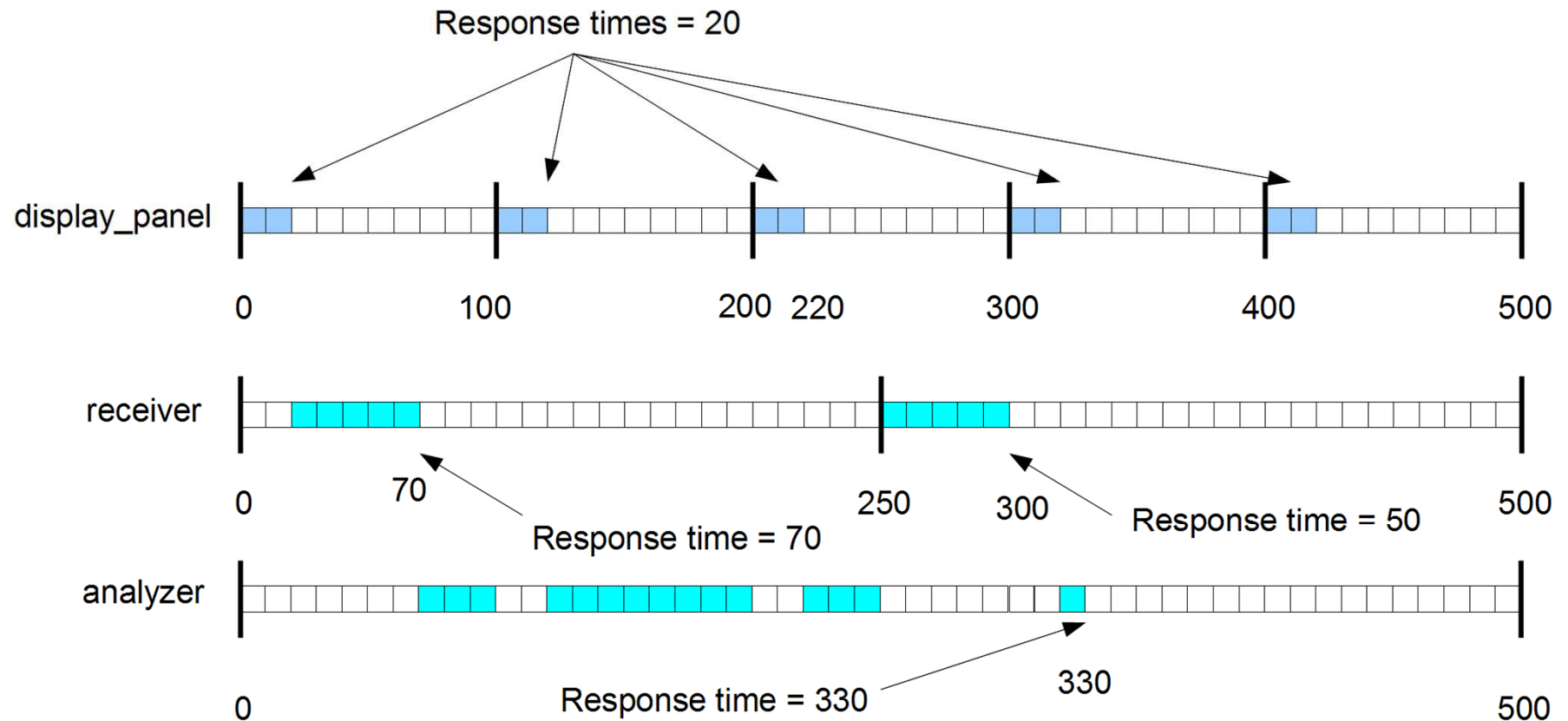
$$w3^5 = C3 + \left\lceil \frac{w3^4}{P1} \right\rceil \cdot C1 + \left\lceil \frac{w3^4}{P2} \right\rceil \cdot C2 = 18 \Rightarrow R3 = 18$$

# Uniprocessor fixed priority scheduling

---

- **Example with the AADL radar case study:**
  - “display\_panel” thread which displays data.  $P=100$ ,  $C=20$ .
  - “receiver” thread which sends data.  $P=250$ ,  $C=50$ .
  - “analyser” thread which analyzes data.  $P=500$ ,  $C=150$ .
  
- **Processor utilization factor test:**
  - $U=20/100+150/500+50/250=0.7$
  - $\text{Bound}=3.(2^{\frac{1}{3}} - 1)=0.779$
  - $U \leq \text{Bound} \Rightarrow$  deadlines will be met.
  
- **Worst case task response time:**  $R_{analyser}=330$ ,  
 $R_{display\_panel}=20$ ,  $R_{receiver}=70$ .
  
- **Run simulations on feasibility interval:**  $[0, \text{LCM}(P_i)] = [0, 500]$ .

# Uniprocessor fixed priority scheduling



# Fixed priority and shared resources

---

- Previous tasks were independent ... does not exist in real life.
  
- **Task dependencies:**
  - Shared resources.
    - E.g. with AADL: threads may wait for AADL protected data component access.
  - Precedencies between tasks.
    - E.g with AADL: threads exchange data by data port connections.



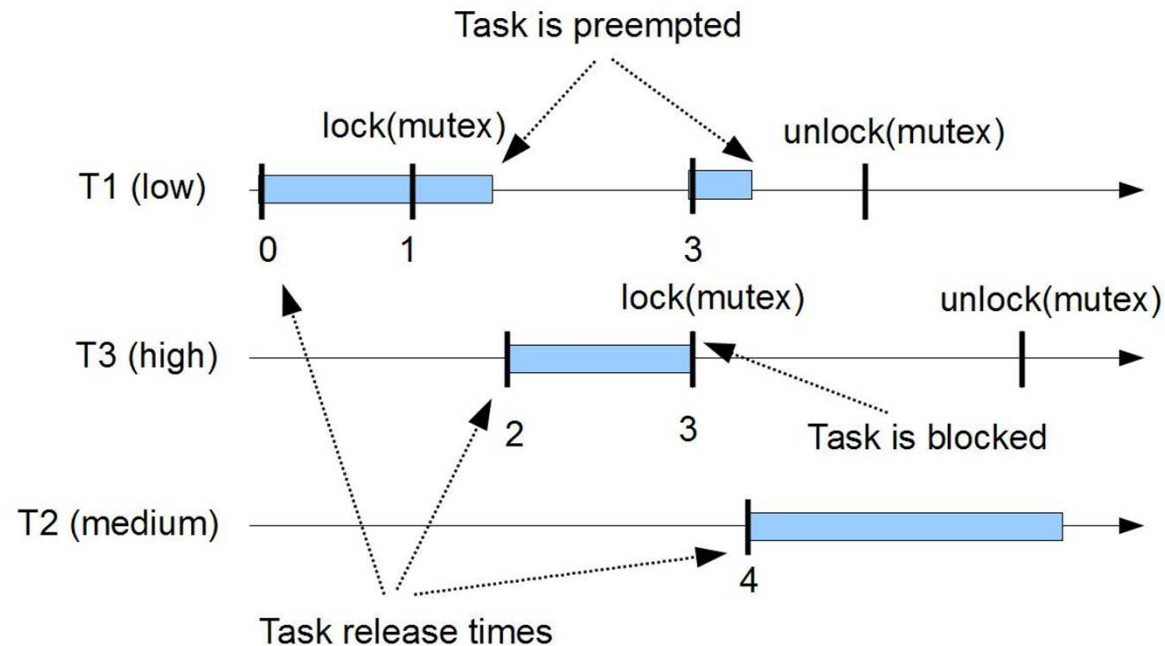
# Fixed priority and shared resources

---

- Shared resources can be modeled by semaphores for scheduling analysis.
- **We use specific semaphores implementing inheritance protocols:**
  - To take care of priority inversion.
  - To compute worst case task waiting time for the access to a shared resource => Blocking time  $B_i$ .
- **Inheritance protocols:**
  - PIP (Priority inheritance protocol), cannot be used with more than one shared resource due to deadlock.
  - PCP (Priority Ceiling Protocol) , implemented in most of real-time operating systems (e.g. VxWorks).
  - Several implementations of PCP exists: OPCP, ICPP, ...

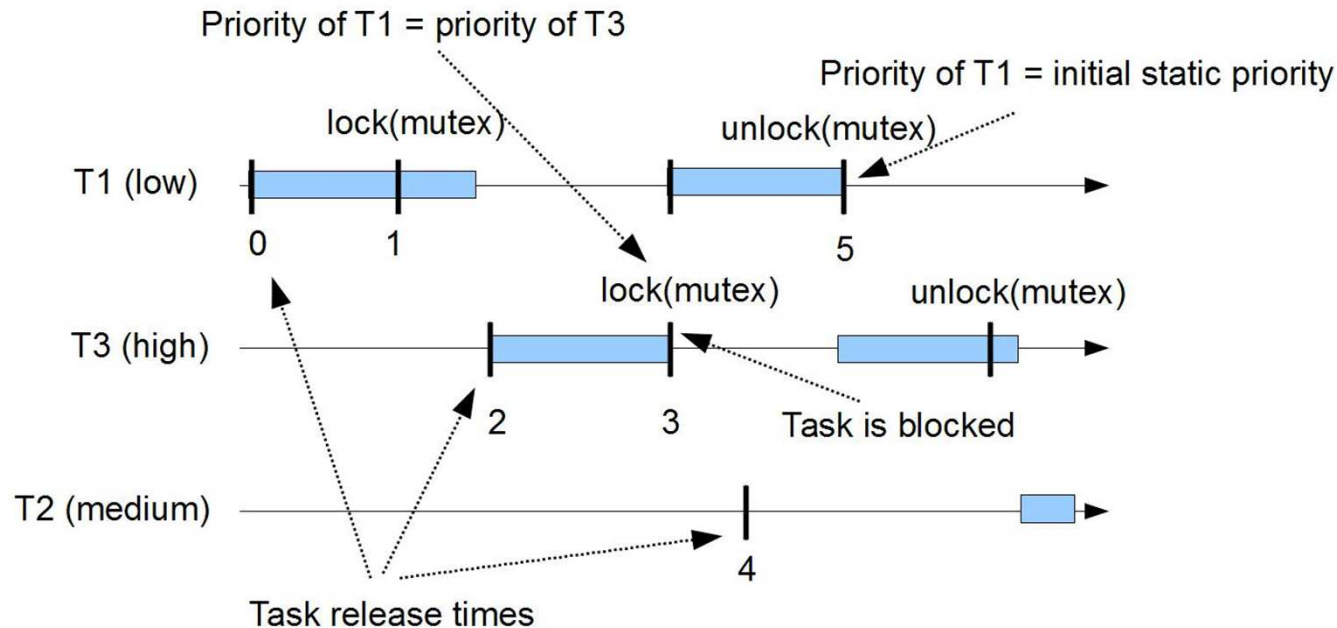
# Fixed priority and shared resources

- **What is priority inversion:** a low priority task blocks a high priority task



- $B_i$  = worst case on the shared resource blocking time.

# Fixed priority and shared resources



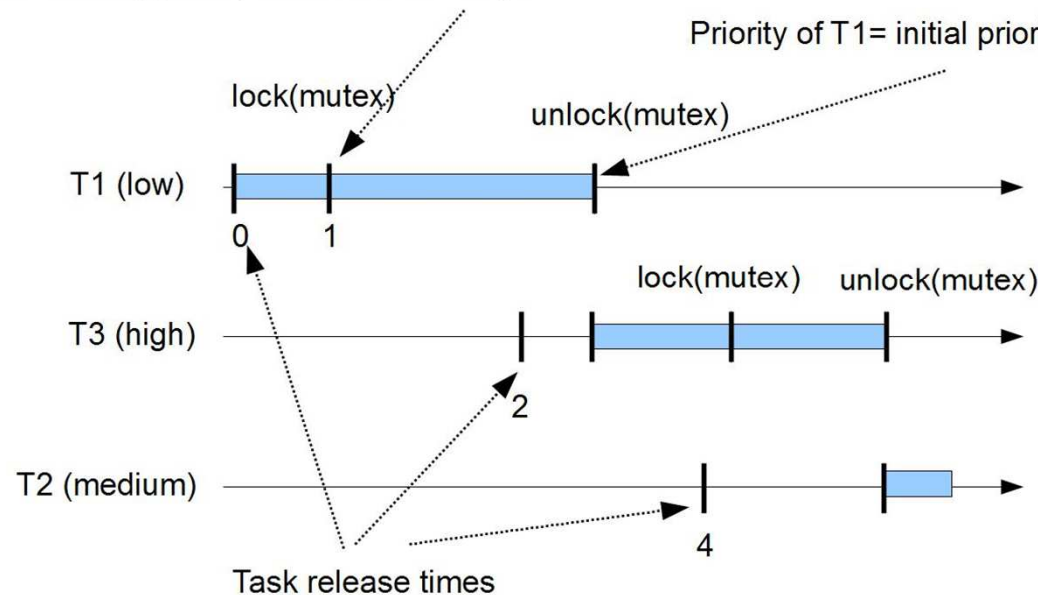
## □ PIP (Priority Inheritance Protocol):

- A task which blocks a higher priority task runs its critical section with the priority level of the blocked task
- Only one shared resource, deadlock otherwise
- $B_i$  = sum of critical section durations of lower priority tasks than  $i$

# Fixed priority and shared resources

Priority of T1 = ceiling priority of « mutex » = high

Priority of T1 = initial priority of T1 = low



## □ ICPP (Immediate Ceiling Priority Protocol):

- Ceiling priority of a resource = maximum fixed priority of the tasks which use it.
- Dynamic task priority = maximum of its own fixed priority and the ceiling priorities of any resources it has locked.
- $B_i$  = longest critical section ; prevent deadlock and reduce blocking

# Fixed priority and shared resources

---

## □ How to take into account $B_i$ (blocking time):

- Processor utilization factor test :

$$\forall i, 1 \leq i \leq n : \sum_{k=1}^{i-1} \frac{C_k}{P_k} + \frac{C_i + B_i}{P_i} \leq i \cdot (2^{\frac{1}{i}} - 1)$$

- Worst case response time :

$$R_i = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{P_j} \right\rceil \cdot C_j$$

# To conclude on scheduling analysis

---

- **Many feasibility tests:** depending on task, processor, scheduler, shared resource, dependencies, multiprocessor, hierarchical, distributed...

$$R_i = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{P_j} \right\rceil \cdot C_j$$

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{P_j} \right\rceil \cdot C_j$$

$$R_i = w_i + J_i$$

$$w_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + J_j}{P_j} \right\rceil \cdot C_j$$

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{P_j} \right\rceil \cdot C_j + \max(C_k \forall k \in hp(i))$$

- **Many assumptions:** require preemptive, fixed priority scheduling, synchronous periodic, independent tasks, deadlines on requests...

**Many feasibility tests... Many assumptions...**

**How to choose them?**

# Summary

---

1. Issues about real-time scheduling analysis: AADL to the rescue
2. Basics on scheduling analysis: fixed-priority scheduling for uniprocessor architectures
3. AADL components/properties to scheduling analysis

# AADL to the rescue ?

---

## □ **Issues:**

1. Ensure all required model elements are given for the analysis
2. Ensure model elements are compliant with analysis requirements/assumptions

## □ **AADL helps for the first issue:**

- AADL as a pivot language between tools. International standard.
- Close to the real-time scheduling theory: real-time scheduling analysis concepts can be found. Ex:
  - Component categories: thread, data, processor
  - Property: Deadline, Fixed Priority, ICPP, Ceiling Priority, ...



# Property sets for scheduling analysis

---

## □ Properties related to processor components:

**Preemptive\_Scheduler** : aadlboolean applies to  
(processor);

**Scheduling\_Protocol**: inherit list of  
Supported\_Scheduling\_Protocols  
**applies to** (virtual processor, processor);

-- *RATE\_MONOTONIC\_PROTOCOL*,

-- *POSIX\_1003\_HIGHEST\_PRIORITY\_FIRST\_PROTOCOL*, ...

# Property sets for scheduling analysis

---

## □ Properties related to the threads/data components:

**Compute\_Execution\_Time**: Time\_Range **applies to** (thread, subprogram, ...);

**Deadline**: **inherit** Time => Period **applies to** (thread, ...);

**Period**: **inherit** Time **applies to** (thread, ...);

**Dispatch\_Protocol**: Supported\_Dispatch\_Protocols **applies to** (thread);

-- *Periodic, Sporadic, Timed, Hybrid, Aperiodic, Background, ...*

**Priority**: **inherit** aadlinteger **applies to** (thread, ..., data);

**Concurrency\_Control\_Protocol**: Supported\_Concurrency\_Control\_Protocols  
**applies to** (data);

-- *None, PCP, ICPP, ...*

# AADL to the rescue ?

---

- **Issues:**

1. Ensure all required model elements are given for the analysis
2. Ensure model elements are compliant with analysis requirements/assumptions

- **And for the second issue?**

# Cheddar : a framework to assess schedulability of AADL models

---

- **Cheddar tool =**
  - + analysis framework (queueing system theory & real-time scheduling theory)
  - + internal ADL (analysis model)
  - + simple analysis model editor
  - + optimization tools
  - + ...
  
- **Two versions:**
  - Open source (Cheddar) : teaching/research, TASTE, OSATE, MOSART, RAMSES, ...
  - Commercial product (AADLInspector) : Ellidiss Tech product.
  
- **Supports** : Ellidiss Tech., Conseil régional de Bretagne, Brest Métropole, Campus France, BPI France

# Cheddar : main analysis features

(see <http://beru.univ-brest.fr/~singhoff/cheddar>)

---

- ❑ **Analysis by scheduling simulations:**
  - Various scheduling policies, uniprocessor, multiprocessor, cache, ...
  - Simulation data analysis
- ❑ **Task schedulability/feasibility tests**
- ❑ **Design space exploration methods**
- ❑ **Task and resource priority assignments**
- ❑ **Partitioning algorithms**
- ❑ **Queueing system theory models/buffer feasibility tests**
- ❑ **Modeling/analysis with task dependencies**

## AADL “design pattern” approach to automatically perform scheduling analysis

---

- ❑ **Let assume we have to evaluate a given architecture model in a design exploration flow.**
  
- ❑ **Problem statement reminder:**
  - Numerous schedulability tests ; how to choose the right one?
  - Numerous assumptions for each schedulability test ; how to enforce them for a given model?
  - How to automatically perform scheduling analysis?

# AADL “design pattern” approach to automatically perform scheduling analysis

---

## ■ **Approach:**

- **Define a set of AADL architectural design patterns of real-time (critical) systems:**
  - = models a typical thread communication or synchronization + a typical execution platform
  - = set of constraints on entities/properties of the model.
- **For each design pattern,** define schedulability tests that can be applied according to their applicability assumptions.
- **Schedulability analysis of an AADL model:**
  1. Check compliancy of his model with one of the design-patterns ... which then gives him which schedulability tests we can apply.
  2. Perform schedulability verification.

# Design pattern compliancy verification

The image shows a screenshot of the Platypus IDE. The top window, titled 'Platypus', displays a real-time system architecture model. The left sidebar shows a tree view with nodes like 'Tamaris', 'DemoPlatypus', 'Express', 'cheddar', 'express', and 'morphtr'. The main editor area shows the following code:

```
DATA;  
#1=PERIODIC_TASK(7, 29, 29, 0, 1, 0);  
#2=PERIODIC_TASK(3, 10, 10, 0, 1, 0);  
#3=PERIODIC_TASK(1, 5, 5, 0, 1, 0);  
ENDSEC;
```

The bottom window, titled 'Simultaneous', displays a feasibility test applicability assumption. The left sidebar shows a tree view with nodes like 'Tamaris', 'express2cheddar', 'cheddar\_data', 'cheddar\_data ma', 'cheddar', 'jcheddar\_data', 'express t', 'RTPatterns', 'morphtr', 'platypus', 'settings', 'Simultaneous', 'interface', 'rules', 'Simultaneous', and 'Period\_Equal\_De'. The main editor area shows the following code:

```
RULE Simultaneous_Release_Time FOR ( Periodic_Task );  
LOCAL  
nbpt : INTEGER := SIZEOF ( Periodic_Task );  
p1 : Periodic_Task := Periodic_Task [ 1 ];  
END_LOCAL;  
WHERE  
(* All tasks share the same release time *)  
r1 : ( nbpt < 2 ) OR  
( SIZEOF ( QUERY ( p < * Periodic_Task |  
p.Release_Time <> p1.Release_Time ) ) = 0 );  
END_RULE;
```

Callouts in the image point to specific parts of the code:

- A green callout points to the top right part of the first window, containing the data definitions for periodic tasks.
- A blue callout points to the bottom right part of the second window, containing the feasibility test applicability assumption.
- A pink callout points to the left part of the second window, containing the result of the model compliancy analysis.

- **Top right part:** real-time system architecture model to verify.
- **Bottom right part:** modeling of a feasibility test applicability assumption.
- **Left part:** result of the model compliancy analysis.



# Example : «Ravenscar» design pattern

---

- **Specification of various design patterns:**
  - **Time-triggered** : sampling data port communication between threads
  - **Ravenscar** : PCP shared data communication between threads
  - **Queued buffer/ARINC653** : producer/consumer synchronization
  - **Black board/ARINC653** : readers/writers synchronization
  - ...
  - **Compositions of design patterns.**
  
- **Ravenscar:** used by TASTE/ESA
  
- **Constraints defining “Ravenscar” to perform the analysis with a given schedulability test:**
  - Constraint 1 : all threads are periodic
  - Constraint 2 : threads start at the same time
  - Constraint 3 : shared data with PCP
  - Constraint n : fixed preemptive priority scheduling + uniprocessor
  - ...

# Example : «Ravenscar» compliant AADL model

---

**thread implementation** receiver.impl

**properties**

Dispatch\_Protocol => Periodic;

Compute\_Execution\_Time => 31 ms .. 50 ms;

Deadline => 250 ms;

Period => 250 ms;

**end** receiver.impl;

**data implementation** target\_position.impl

**properties**

Concurrency\_Control\_Protocol

=> PRIORITY\_CEILING\_PROTOCOL;

**end** target\_position.impl;

**process implementation** processing.others

**subcomponents**

receiver : **thread** receiver.impl;

analyzer : **thread** analyzer.impl;

target : **data** target\_position.impl;

...

**processor implementation** leon2

**properties**

Scheduling\_Protocol =>

RATE\_MONOTONIC\_PROTOCOL;

Preemptive\_Scheduler => true;

**end** leon2;

**system implementation** radar.simple

**subcomponents**

main : **process** processing.others;

cpu : **processor** leon2;

...

# Demos, practical labs

- Scheduling analysis of the radar example with AADLInspector & OSATE/Cheddar

The screenshot shows the AADLInspector interface. On the left, the code for 'arincsimple2' is displayed, including package declarations, system definitions, and processor implementations. On the right, the 'Schedulability' tab is active, showing a table of test results for various entities. Below the table, a Gantt chart visualizes the execution of tasks on the 'cpu' processor over time.

test	entity	value
Task response time computed from simulation	cpu	No deadline miss
Number of preemptions	cpu	4
Number of context switches	cpu	74
Task response time computed from simulation	cpu.partition1_pr.T	worst = 5, best = 5, average = 5
Task response time computed from simulation	cpu.partition1_pr.T	worst = 15, best = 15, average = 15
Task response time computed from simulation	cpu.partition2_pr.T	worst = 15, best = 15, average = 15
Set priorities according to Rate Monotonic	cpu	
Set priorities according to Deadline Monotonic	cpu	

The screenshot shows the Cheddar simulator interface. It displays a Gantt chart for four tasks (T1, T2, T3, T4) over time. Below the chart, a text box provides scheduling statistics for the processor 'arinc'.

**Scheduling simulation, Processor arinc :**

- Number of preemptions : 760
- Number of context switches : 3205
- Task response time computed from simulation :
  - T1 => 6/worst 6/best 6.00000/average
  - T2 => 56/worst 35/best 46.81667/average
  - T3 => 10/worst 4/best 6.00000/average
  - T4 => 1/worst 1/best 1.00000/average
- No deadline missed in the computed scheduling : the task set seems to be schedulable.

# Conclusion



# To summarize

---

## □ **We introduced the concepts of AADL**

- Architectural description language
- Patterns for scheduling analysis

## □ **Not discussed today:**

- Code generation => Ocarina, J. Hugues/ISAE
- Reliability analysis using Error Modeling Annex => P. Feiler CMU/SEI
- Modeling of IMA systems => L. Pautet and E. Borde/Télécom Paris
- Network models & analysis => A. Khoroshilov/ISPRAS
- Multiprocessor support & scheduling analysis => S. Rubini and F. Singhoff/Lab-STICC
- Formal methods => B. Larson/KS Univ., J.P. Talpin/INRIA, M. Filali/IRIT
- Design exploration => L. Pautet and E. Borde/Télécom Paris , J. Hugues/ISAE, L. Lemarchand and F. Singhoff/Lab-STICC
- and much more !