
# AADL: Architecture Analysis & Design Language

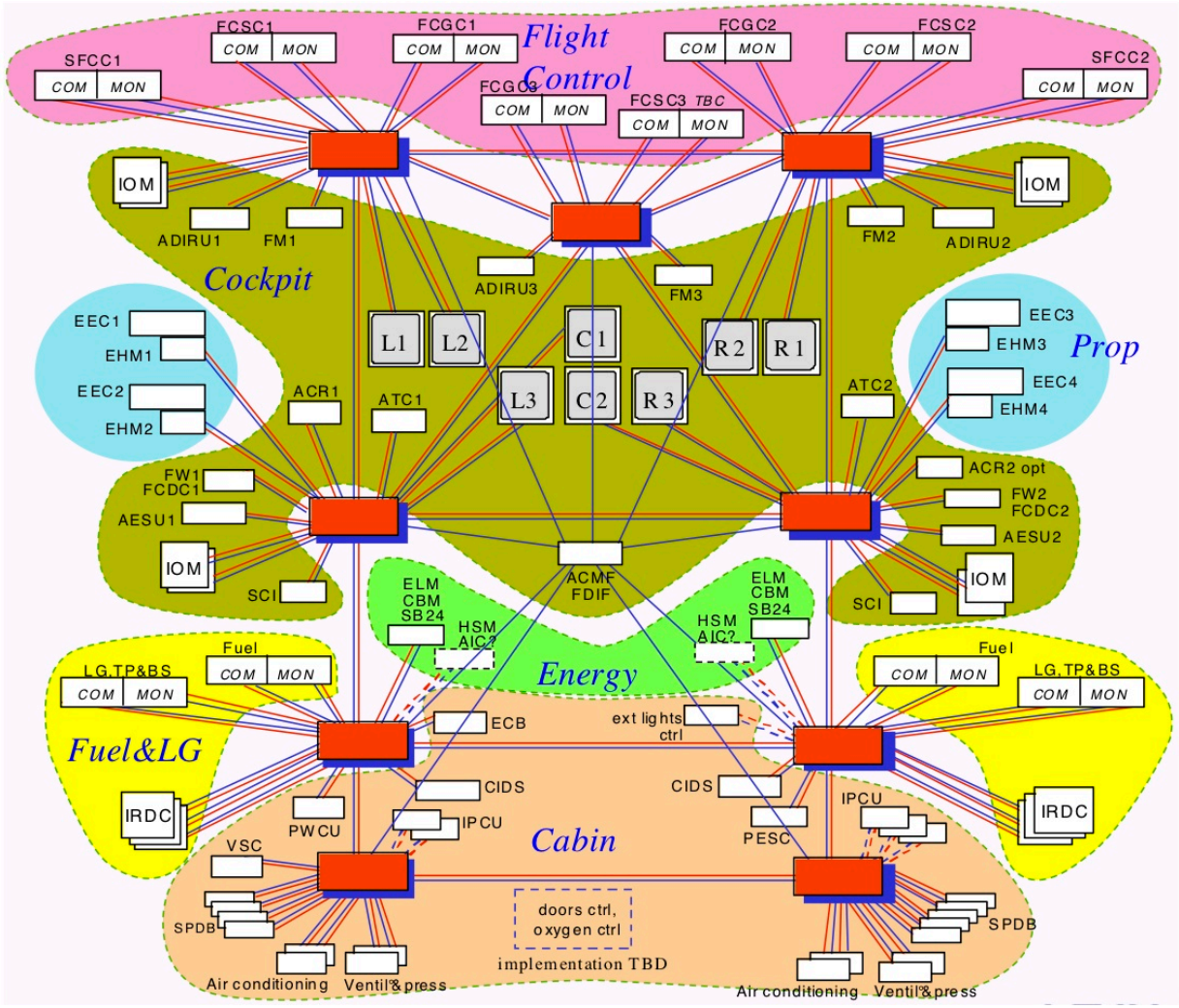
***Analyse temporelle***

Etienne Borde
etienne.borde@telecom-paristech.fr

# Objectifs du cours

- Savoir modéliser un système embarqué en AADL pour
  - ⌘ Faire des analyses de temps de réponse des tâches
  - ⌘ Faire des analyses de temps de latence et de gigue
- Savoir calculer le temps de latence d'une donnée
- Savoir calculer la gigue associé au temps de latence d'une donnée
- Connaître les limitations de cette technique

# Exemple d'architecture avionique




*Comment, et pourquoi modéliser une telle architecture?*


# Plan du cours

1. Présentation du langage AADL
2. Modèles AADL pour le calcul de temps de réponse
3. Définition du temps de latence et de sa gigue
4. Modèles AADL pour le calcul de temps de latence


# Les principes de base d'AADL

# AADL: objectif principal et moyens

- Faciliter la conception (*structuration et analyse d'architectures*) des systèmes temps-réels distribués
  - ⌘ Définit une sémantique, standardisée, aussi précise (et concrète) que possible pour l'ensemble des éléments du langage
    - La sémantique est décrite en langage naturelle dans un standard, ~340 pages hors annexes
  - ⌘ Ne pouvant couvrir l'ensemble des exigences de conception du domaine, AADL propose des mécanismes d'extension (*i.e.* langage de propriétés et annexes)
  - ⌘ Propose trois niveau de modélisation:
    1. Système
    2. Logiciel
    3. Matériel

# AADL: objectifs détaillés (mais partiels)

- Analyse de systèmes temps réel
  - ⌘ Temps de réponse des tâches
  - ⌘ Temps de transmission des données (temps de latence, gigue)
  - ⌘ Disponibilité, fiabilité, …
- Modélisation de code legacy
  - ⌘ Représentation du code
  - ⌘ Représentation des données
- Génération automatique de code
  - ⌘ Diversité des langages de programmation (Java, C, Ada, …)
  - ⌘ Diversité des systèmes d'exploitations avec leurs API (RT-POSIX, FreeRTOS, VxWorks, ARINC653, OSEK…)

# Historique

- Anciennement « Avionics Architecture Description Language »
- Evolution de MetaH, qui était développé par Honeywell
- Plusieurs représentations
  - représentation textuelle
    - pour contrôler tous les détails du système
  - représentation graphique
    - convenable pour avoir une vision globale du système
- Version 1.0 publiée en 2004, version 2.0 a été publiée fin 2009

# Usage des modèles d'architectures dans l'industrie?

- Surtout UML, mais DSLs (Domain Specific languages) dans le domaine de l'embarqué
  - Chaque société à son propre ADL ou presque…
  - Nous utilisons AADL pour illustrer les principes généraux de ce type de langages

- AADL surtout utilisé dans des projets de R&D mais supporté par des outils industriels avec un intérêt croissant

- Objectifs:
  - Spécification,
  - Documentation,
  - Conception,
  - Analyse,
  - Génération de code


# Les composants AADL,
# et leur composition

# Caractéristiques générales

- Langages de description d'architecture: les ***composants*** sont les éléments de base du langage.
- En AADL, les composants appartiennent à une Catégorie (Process, thread, data, processor, etc.)
- La déclaration d'un composants est structurée en
  - Déclaration du type de composant: vise à définir ses interfaces de communications avec d'autres composant (ports, access points, …)
  - Déclaration de l'implémentation de composant: vise à définir la structure interne du composants (sous-composant, code, spec. comportementale, etc…)
- Pour 1 Catégorie, on peut déclarer plusieurs types; pour 1 type on peut déclarer plusieurs implémentations
- Des propriétés (prédéfinies ou définies par le concepteur) peuvent être associées à chaque élément de modélisation (type de composant, implémentation, sous-composant, ports, connections,…)
- Langage descriptif: les composants peuvent être spécifiés dans n'importe quel ordre
- Langage non sensitif à la casse

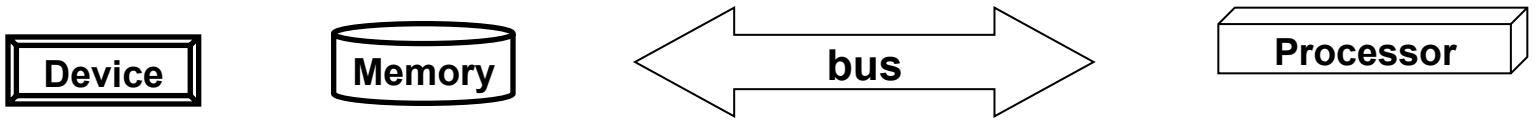
# Catégories de composants

- Éléments de base d'une description architecturale: toute déclaration de composant AADL (type ou implémentation) doit correspond à une catégorie de composant

- Plusieurs ensembles de catégories
  - Matériel (*execution platform components*)
  - Logiciel (*software components*)
  - système (*system composition*)

# Description de la plate-forme d'exécution

- Plusieurs catégories
  - processor : unité de calcul + ordonnanceur
  - memory : composant de stockage (disque dur, mémoire vive, RAM, cache, etc.)
  - bus : lien physique de communication (réseau, etc.)
  - device : interfaces physique/logique du système avec l'environnement (capteur/actionneur/pilote de périphérique)
- Attention:
  - un processor modélise processeur + noyau contenant entre-autres un ordonnanceur.
  - Un device sert typiquement à modéliser un capteur + le pilote de ce capteur (composant matériel et logiciel)

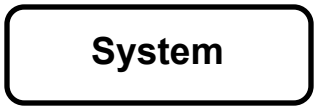


# Description du logiciel

- Plusieurs catégories
  - ⌘ thread : tâche qui exécute des fonctions du système (pas forcément un thread du noyaux…)
  - ⌘ data : type de données abstrait (pas forcément de correspondance avec un type de donnée en programmation)
  - ⌘ process : processus, un espace mémoire d'adressage logique pour l'exécution des threads qu'il contient
  - ⌘ subprogram : procédure, comme pour les langages de programmation. N'as pas de valeur de retour
- Un process doit contenir au moins un thread.

# Description système

- Permet de structurer la description (matériel+logiciel)
- Un implémentation de composant de catégorie système peut contenir des sous-composant de catégorie:
  - ⌘ system
  - ⌘ processor, memory, device, bus
  - ⌘ process, data
- MAIS PAS de catégorie:
  - ⌘ thread
  - ⌘ thread group
  - ⌘ subprogram



# Composition de composants AADL

- Une implémentation de composant peut avoir des sous-composants
- Une modélisation AADL est donc une arborescence de composants, commençant par un composant racine de la catégorie system.

| Implémentation de catégorie | Peut contenir des sous-composant de catégorie |
|---|---|
| data | data |
| thread | data |
| thread group | data, thread |
| process | thread, |
| processor | memory |
| memory | memory |
| system | tous sauf subprogram, thread et thread group |

# Eléments de syntaxe (package)

```
----------------------------------------
-- Package describing the architecture
-- of the line follower robot
----------------------------------------

package nxt_use_case -- beginning of package declaration
public -- beginning of public (i.e. visible) package section

with Data_Model; -- gives visibility to existing declarations in the
                 -- public section of a package nammed Data_Model (if any),
                 -- or to a property set nammed Data_Model (if any)



-- components declaration will be added here




end nxt_use_case; -- end of package declaration
```

# Eléments de syntaxe (component type)

```
package nxt_use_case
public

with Data_Model;

system nxt -- beginning of declaration of component type nxt, of category system
end nxt;   -- end of declaration of component type nxt

processor arm
end arm; -- declaration of component type arm, of category processor

process proc
end Proc; -- declaration of component type proc, of category process


end nxt_use_case;
```

# Eléments de syntaxe (component implementation)

```
package nxt_use_case
public

system nxt
end nxt;

system implementation nxt.Impl
end nxt.Impl; -- declaration of component implementation nxt.Impl,
              -- of type nxt (declared above), of category system

processor arm
end arm;

processor implementation arm.v7
end arm.v7; -- declaration of component implementation arm.v7,
            -- of type arm (declared above), of category system

end nxt_use_case;
```

# Eléments de syntaxe (sous-composants)

```
package nxt_use_case
public

system nxt
end nxt;

system implementation nxt.Impl
  subcomponents -- subcomponents section in nxt.Impl
    cpu1: processor arm.v7;
    cpu2: processor arm.v7;
    proc1: process proc.impl;
    proc2: process proc.impl;
end nxt.Impl;

-- the declaration of arm.v7 and proc.impl have been removed for the sake of
-- brevity; see slides above to retreive these declarations

end nxt_use_case;
```

# Notes sur la syntaxe vue précédemment

- Commentaires commencent par --

- Déclaraction d'un package pkg_id:

```
package pkg_id
public
…
end pkg_id;
```

- Déclaration d'un type de composant cpt_id

```
process cpt_id
end cpt_id;
```

- Déclaration d'une implémentation de composant cpt_id.impl_id, avec un sous composant subcpt_id.

```
process implementation cpt_id.impl_id
 subcomponents
   subcpt_id: thread other_cpt_id; -- ou other_cpt_id.other_impl_id
end cpt_id.impl_id;
```


# Le rôle des propriétés dans un modèle AADL

# Les propriétés en AADL: principes de base

- Une propriété permet d'associer une valeur d'un certain type à un élément du modèle.
  - Le standard AADL définit un ensemble de ***propriétés standard***
  - il est possible de définir de nouvelles propriétés dans des ensembles de propriétés (**property sets**)
    - un langage complémentaire à celui présenté jusque là

- Les propriétés peuvent être associées à quasiment tous les éléments d'une description d'architecture en AADL

- Dans le langage de définition des propriétés, on peut contraindre l'applicabilité d'une propriété à un ensemble d'éléments (p.ex. ceux de catégorie processor uniquement)

# Les propriétés en AADL: premiers éléments de syntaxe

- Une propriété peut être du type
  - un booléen : `aadlboolean`
  - un entier : `aadlinteger`
  - un réel : `aadlreal`
  - une chaîne de caractères : `aadlstring`
  - une énumération : `enumeration`
  - une catégorie d'élément : `classifier` (composant, connexion, etc.)
  - une référence à un element: `reference` (composant…)
  - une plage de valeurs : `list of …`
  - A metrics unit : `unit`
- On peut
  - définir des types de propriété en réutilisant des types existants
  - associer une valeur par défaut à une propriété
- `applies to` spécifie à quels types d'éléments du modèle la propriété peut s'appliquer

# Eléments de syntaxe: déclaration d'un ensemble de propriétés (property set)

```
-------------------------------
-- declaration of a property set
-- named propertyset_id
-------------------------------


property set propertyset_id is

  -- declaration of properties should be added here

end propertyset_id;
```

# Propriétés en lien avec RTA

- Les composants AADL concernés sont
  - Les threads, composant « principaux » en AADL
  - Les processeurs car ils contiennent l'ordonnanceur
- L'ordonnancement est définie grâce à des propriétés prédéfinies
  - *Dispatch_protocol*, le thread est
    - ***periodic***, le thread est réveillé périodiquement
    - ***sporadic***, le thread est réveillé sur réception de messages, avec un délais minimale entre deux réveils
    - ***aperiodic,*** le thread est réveillé sur réception de messages
    - ***timed,*** le thread est réveillé ***soit*** sur réception de messages ***soit*** sur échéance temporelle (timer réinitialisé sur réception de message)
    - ***Hybrid,*** le thread est réveillé ***à la fois*** sur réception de messages ***et*** sur échéance temporelle
  - *Compute_execution_time* représente le temps d'exécution d'un thread ou d'un sous-programme.
  - *Priority* permet d'associer une valeur de priorité à un thread.
  - *Deadline* permet d'associer une valeur d'échéance à un thread.
  - *Scheduling_Protocol* précise la politique d'ordonnancement associé à un processeur.

# Eléments de syntaxe: déclaration de propriétés (propriétés utilisées par la suite)

```
-------------------------------------------------------
-- declaration of a subset of standardized properties
-- in property set named Timing_Properties
-------------------------------------------------------

property set Timing_Properties is

  Time: type aadlinteger 0 ps .. Max_Time units Time_Units;

  Time_Range: type range of Time;

  Period: Time applies to
    (thread, thread group, process, system, device, virtual processor,
    virtual bus, bus);

  Compute_Execution_Time: Time_Range
    applies to (thread, device, subprogram, event port, event data port);

  Deadline: Time => Period applies to
    (thread, thread group, process, system, device, virtual processor);

end Timing_Properties;
```

Identifiant de la propriété ou du type

# Eléments de syntaxe: déclaration de propriétés (propriétés utilisées par la suite)

```
-------------------------------------------------------
-- declaration of a subset of standardized properties
-- in property set named Timing_Properties
-------------------------------------------------------

property set Timing_Properties is

  Time: type aadlinteger 0 ps .. Max_Time units Time_Units;

  Time_Range: type range of Time;

  Period: Time applies to
    (thread, thread group, process, system, device, virtual processor,
    virtual bus, bus);

  Compute_Execution_Time: Time_Range
    applies to (thread, device, subprogram, event port, event data port);

  Deadline: Time => Period applies to
    (thread, thread group, process, system, device, virtual processor);

end Timing_Properties;
```

Il s'agit d'un type de valeur

# Eléments de syntaxe: déclaration de propriétés (propriétés utilisées par la suite)

```
-------------------------------------------------------
-- declaration of a subset of standardized properties
-- in property set named Timing_Properties
---------------------------------------
property set Timing_Properties is

  Time: type aadlinteger 0 ps .. Max_Time units Time_Units;

  Time_Range: type range of Time;

  Period: Time applies to
    (thread, thread group, process, system, device, virtual processor,
    virtual bus, bus);

  Compute_Execution_Time: Time_Range
    applies to (thread, device, subprogram, event port, event data port);

  Deadline: Time => Period applies to
    (thread, thread group, process, system, device, virtual processor);

end Timing_Properties;
```

Les valeurs admises pour ce type sont des entiers entre 0 et Max_Time

# Eléments de syntaxe: déclaration de propriétés (propriétés utilisées par la suite)

```
-------------------------------------------------------
-- declaration of a subset of standardized properties
-- in property set named Timing_Properties
-------------------------------------------------------

property set Timing_Properties is

  Time: type aadlinteger 0 ps .. Max_Time units Time_Units;

  Time_Range: type range of Time;

  Period: Time applies to
    (thread, thread group, process, system, device, virtual processor,
    virtual bus, bus);

  Compute_Execution_Time: Time_Range
    applies to (thread, device, subprogram, event port, event data port);

  Deadline: Time => Period applies to
    (thread, thread group, process, system, device, virtual processor);

end Timing_Properties;
```

Les valeurs admises pour ce type ont une unité Time_Units (défini dans un autre PS)

# Eléments de syntaxe: déclaration de propriétés (propriétés utilisées par la suite)

```
-------------------------------------------------------
-- declaration of a subset of standardized properties
-- in property set named Timing_Properties
-------------------------------------------------------

property set Timing_Properties is

  Time: type aadlinteger 0 ps .. Max_Time units Time_Units;

  Time_Range: type range of Time;

  Period: Time applies to
    (thread, thread group, process, system, device, virtual processor,
    virtual bus, bus);

  Compute_Execution_Time: Time_Range
    applies to (thread, device, subprogram, event port, event data port);

  Deadline: Time => Period applies to
    (thread, thread group, process, system, device, virtual processor);

end Timing_Properties;
```

Time_Range est un type admettant comme valeur des intervals de temps

# Eléments de syntaxe: déclaration de propriétés (propriétés utilisées par la suite)

```
-------------------------------------------------------
-- declaration of a subset of standardized properties
-- in property set named Timing_Properties
-------------------------------------------------------

property set Timing_Properties is

  Time: type aadlinteger 0 ps .. Max_Time units Time_Units;

  Time_Range: type range of Time;

  Period: Time applies to
    (thread, thread group, process, system, device, virtual processor,
    virtual bus, bus);

  Compute_Execution_Time: Time_Range
    applies to (thread, device, subprogram, event port, event data port);

  Deadline: Time => Period applies to
    (thread, thread group, process, system, device, virtual processor);

end Timing_Properties;
```

Period est un identifiant de propriété dont le type est Time

# Eléments de syntaxe: déclaration de propriétés (propriétés utilisées par la suite)

```
-------------------------------------------------------
-- declaration of a subset of standardized properties
-- in property set named Timing_Properties
-------------------------------------------------------

property set Timing_Properties is

  Time: type aadlinteger 0 ps .. Max_Time units Time_Units;

  Time_Range: type range of Time;

  Period: Time applies to
    (thread, thread group, process, system, device, virtual processor,
    virtual bus, bus);

  Compute_Execution_Time: Time_Range
    applies to (thread, device, subprogram, event port, event data port);

  Deadline: Time => Period applies to
    (thread, thread group, process, system, device, virtual processor);

end Timing_Properties;
```

Compute_Execution_Time est un identifiant de propriété
dont le type est un intervalle de temps Time_Range

# Eléments de syntaxe: déclaration de propriétés (propriétés utilisées par la suite)

```
-------------------------------------------------------
-- declaration of a subset of standardized properties
-- in property set named Timing_Properties
-------------------------------------------------------

property set Timing_Properties is

  Time: type aadlinteger 0 ps .. Max_Time units Time_Units;

  Time_Range: type range of Time;

  Period: Time applies to
    (thread, thread group, process, system, device, virtual processor,
    virtual bus, bus);

  Compute_Execution_Time: Time_Range
    applies to (thread, device, subprogram, event port, event data port);

  Deadline: Time => Period applies to
    (thread, thread group, process, system, device, virtual processor);

end Timing_Properties;
```

La valeur par défaut pour la deadline est la valeur associée à la propriété Period

# Eléments de syntaxe: déclaration de propriétés (propriétés utilisées par la suite)

```
-------------------------------------------------------
-- declaration of a subset of standardized properties
-- in property set named AADL_Project
-------------------------------------------------------

property set AADL_Project is

  Supported_Dispatch_Protocols: type enumeration
     (Periodic, Sporadic, Aperiodic, Timed, Hybrid, Background);

  -- Supported_Dispatch_Protocols is a property type, defined as an enumeration.
  -- Enumerated values are Periodic, Sporadic, … ; explained later in the lecture.

  Supported_Scheduling_Protocols: type enumeration
    (Static, Round_Robin_Protocol, POSIX_1003_HIGHEST_PRIORITY_FIRST_PROTOCOL,
    FixedTimeline, Cooperative, RMS, DMS, EDF, SporadicServer, SlackServer,
    ARINC653);

  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_Project;
```

# Eléments de syntaxe: déclaration de propriétés (propriétés utilisées par la suite)

```
-------------------------------------------------------
-- declaration of a subset of standardized properties
-- in property set named AADL_Project
-------------------------------------------------------

property set AADL_Project is

  Supported_Dispatch_Protocols: type enumeration
     (Periodic, Sporadic, Aperiodic, Timed, Hybrid, Background);

  -- Supported_Dispatch_Protocols is a property type, defined as an enumeration.
  -- Enumerated values are Periodic, Sporadic, … ;

  Supported_Scheduling_Protocols: type enumeration
    (Static, Round_Robin_Protocol, POSIX_1003_HIGHEST_PRIORITY_FIRST_PROTOCOL,
    FixedTimeline, Cooperative, RMS, DMS, EDF, SporadicServer, SlackServer,
    ARINC653);

  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_Project;
```

Correspond à FPS dans cours RTA

Voir cours RTA

# Eléments de syntaxe: déclaration de propriétés (propriétés utilisées par la suite)

```
-------------------------------------------------------
-- declaration of a subset of standardized properties
-- in property set named Thread_Properties
-------------------------------------------------------


property set Thread_Properties is

  Priority: aadlinteger applies to
            (thread, thread group, process, system, device, data, data access);

  -- Priority is the property identifier;
  -- Values of this property should be integers;
  -- This property can be applied to components of category thread, process,
     system…

  Dispatch_Protocol: Supported_Dispatch_Protocols
        applies to (thread, device, virtual processor);

end Thread_Properties;
```

# Eléments de syntaxe: déclaration de propriétés (propriétés utilisées par la suite)

```
-------------------------------------------------------
-- declaration of a subset of standardized properties
-- in property set named Deployment_Properties
-------------------------------------------------------


property set Deployment_Properties is

  Scheduling_Protocol: inherit list of Supported_Scheduling_Protocols
        applies to (virtual processor, processor, system);

end Deployment_Properties;
```

# Association de propriétés

- Pour associer une propriété à un composant, on peut :
  1. L'associer à la déclaration du type, et/ou
  2. L'associer à la déclaration de l'implémentation, et/ou
  3. L'associer à la déclaration d'un sous-composant, et/ou
  4. L'associer à partir d'un composant parent.

- Règle de « priorité »: Si on utilise les quatre possibilités, la valeur associée avec la possibilité 4 prévaut sur 3, qui prévaut sur 2, qui prévaut sur 1…

# Exemples d'associations de propriété (scheduling protocol)

```
package nxt_use_case
public

system nxt end nxt;

system implementation nxt.Impl
  subcomponents
    cpu1: processor arm.v7 {Scheduling_Protocol => (ARINC653);};
  properties
    Scheduling_Protocol => (RMS) applies to cpu1;
end nxt.Impl;

processor arm
  properties
    Scheduling_Protocol => (Round_Robin_Protocol);
end arm;

processor implementation arm.v7
  properties
    Scheduling_Protocol => (POSIX_1003_HIGHEST_PRIORITY_FIRST_PROTOCOL);
end arm.v7;
…
```

Politique d'ordo = ?

# Exemples d'associations de propriété (scheduling protocol)

```
package nxt_use_case
public

system nxt end nxt;

system implementation nxt.Impl
  subcomponents
    cpu1: processor arm.v7 {Scheduling_Protocol => (ARINC653);};
  properties
    Scheduling_Protocol => (RMS) applies to cpu1;
end nxt.Impl;

processor arm
  properties
    Scheduling_Protocol => (Round_Robin_Protocol);
end arm;

processor implementation arm.v7
  properties
    Scheduling_Protocol => (POSIX_1003_HIGHEST_PRIORITY_FIRST_PROTOCOL);
end arm.v7;
…
```

Politique d'ordo = RMS (voir règle énoncée plus haut)!

# RTA…
# Last but not least … Actual_Processor_Binding

- On a vu
  - d'une part, l'architecture du logiciel (process+thread+propriétés)
  - d'autre part, l'architecture du matériel (basique pour le moment: un ensemble de processeurs+propriétés)

- Il faut allouer les fonctions du logiciel aux ressources du matériel
  - Propriété Actual_Processor_Binding
  - Usage:

```
SYSTEM IMPLEMENTATION synchronous.others
  SUBCOMPONENTS
    my_platform : PROCESSOR CPU;
    my_process : PROCESS my_process.impl;
  PROPERTIES
    Actual_Processor_Binding => ( reference(my_platform) )
           applies to my_process;
END synchronous.others;
```

# Last but not least… Actual_Processor_Binding

- On a vu
  - d'une part, l'architecture du logiciel (process+thread+propriétés)
  - d'autre part, l'architecture du matériel (basique pour le moment: un ensemble de processeurs+propriétés)

- Il faut allouer les fonctions du logiciel aux ressources du matériel
  - Propriété Actual_Processor_Binding
  - Usage:

```
SYSTEM IMPLEMENTATION synchronous.others
  SUBCOMPONENTS
    my_platform : PROCESSOR CPU;
    my_process : PROCESS my_process.impl;
  PROPERTIES
    Actual_Processor_Binding => ( reference(my_platform) )
          applies to my_process;
END synchronous.others;
```

Spécifie que my_process est alloué à my_platform


# Utilisation de AADL pour automatiser le temps de réponse

# Exemple (1/2)

```
PACKAGE synchronous_Pkg
PUBLIC
WITH Base_Types;

SYSTEM synchronous
END synchronous;

SYSTEM IMPLEMENTATION synchronous.others
  SUBCOMPONENTS
    my_platform : PROCESSOR CPU;
    my_process : PROCESS my_process.impl;
PROPERTIES
Actual_Processor_Binding =>
         ( reference(my_platform) )
           applies to my_process;
END synchronous.others;
```

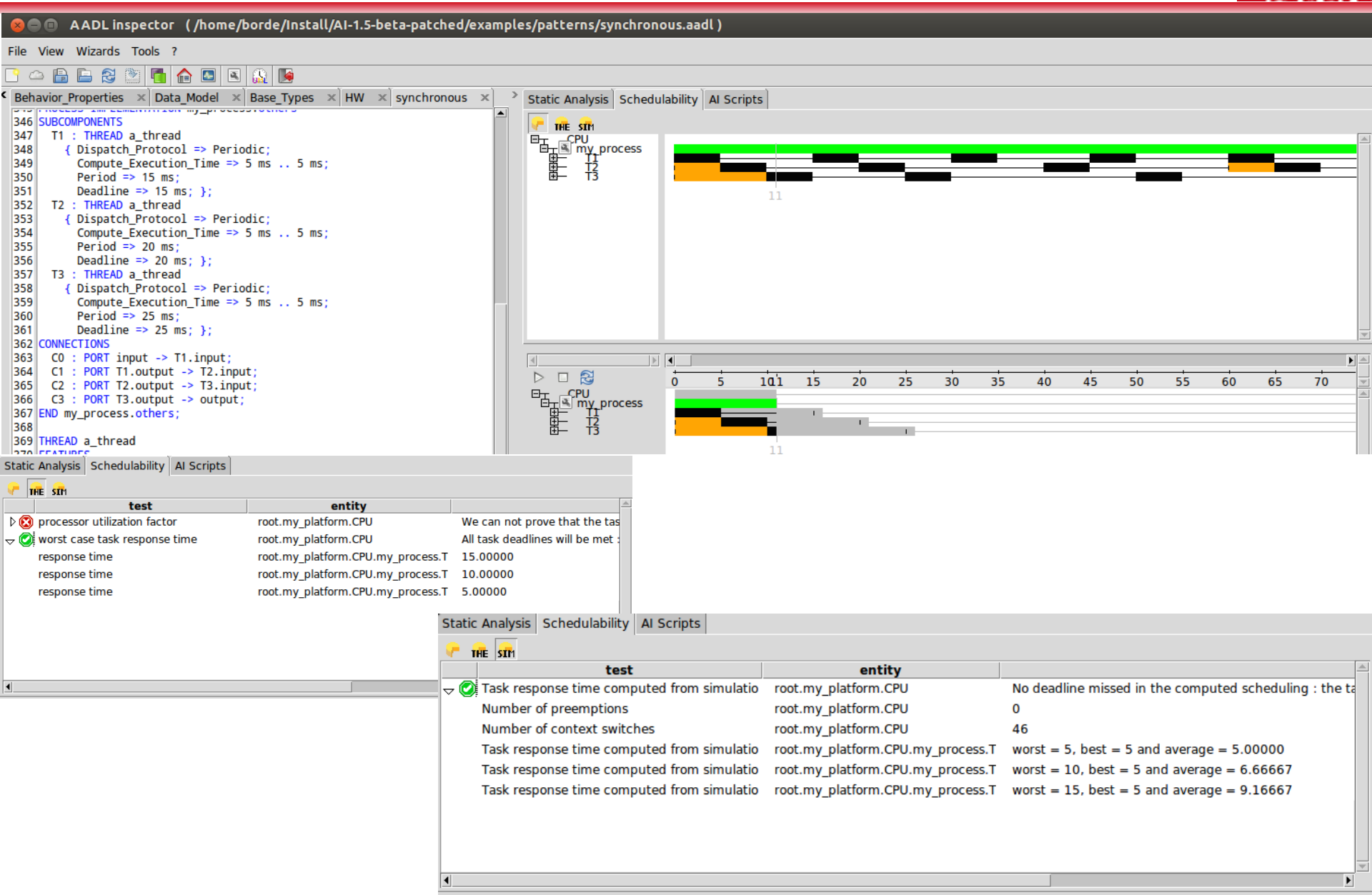
```
PROCESSOR CPU
PROPERTIES
    Scheduling_Protocol => (RMS);
END CPU;


PROCESS my_process
END my_process;
```

# Exemple (2/2)

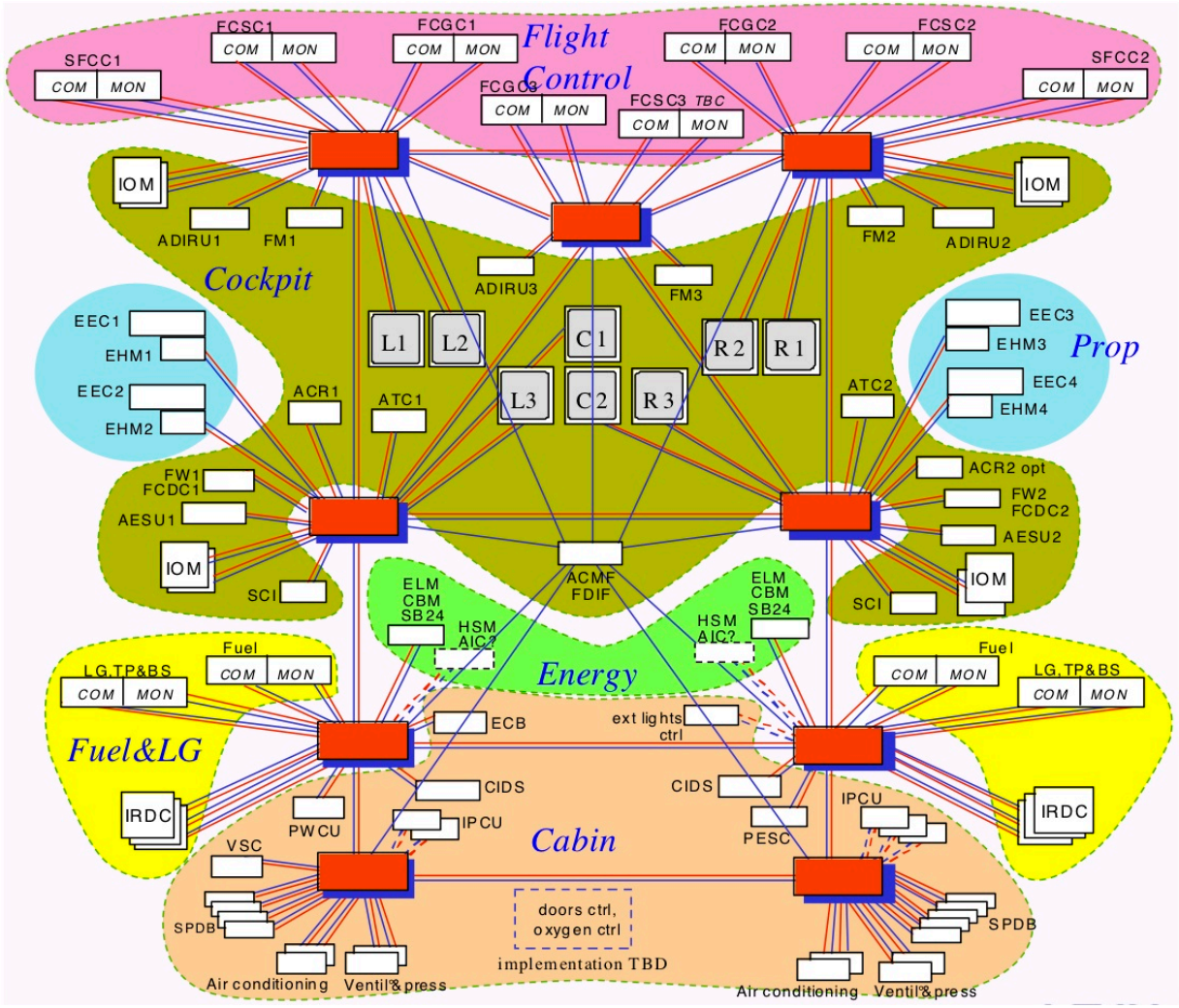
```
PROCESS IMPLEMENTATION my_process.impl
SUBCOMPONENTS
T1 : THREAD a_thread
   { Dispatch_Protocol => Periodic;
     Compute_Execution_Time=>5 ms..5 ms;
     Period => 15 ms;
     Deadline => 15 ms; };
T2 : THREAD a_thread
   { Dispatch_Protocol => Periodic;
     Compute_Execution_Time=>5 ms..5 ms;
     Period => 20 ms;
     Deadline => 20 ms; };
T3 : THREAD a_thread
   { Dispatch_Protocol => Periodic;
     Compute_Execution_Time=>5 ms..5 ms;
     Period => 25 ms;
     Deadline => 25 ms; };
END my_process.impl;
```

```
THREAD a_thread
END a_thread;

END synchronous_Pkg;
```

# Exploitation avec AADL Inspector



# Rappel: architecture avionique



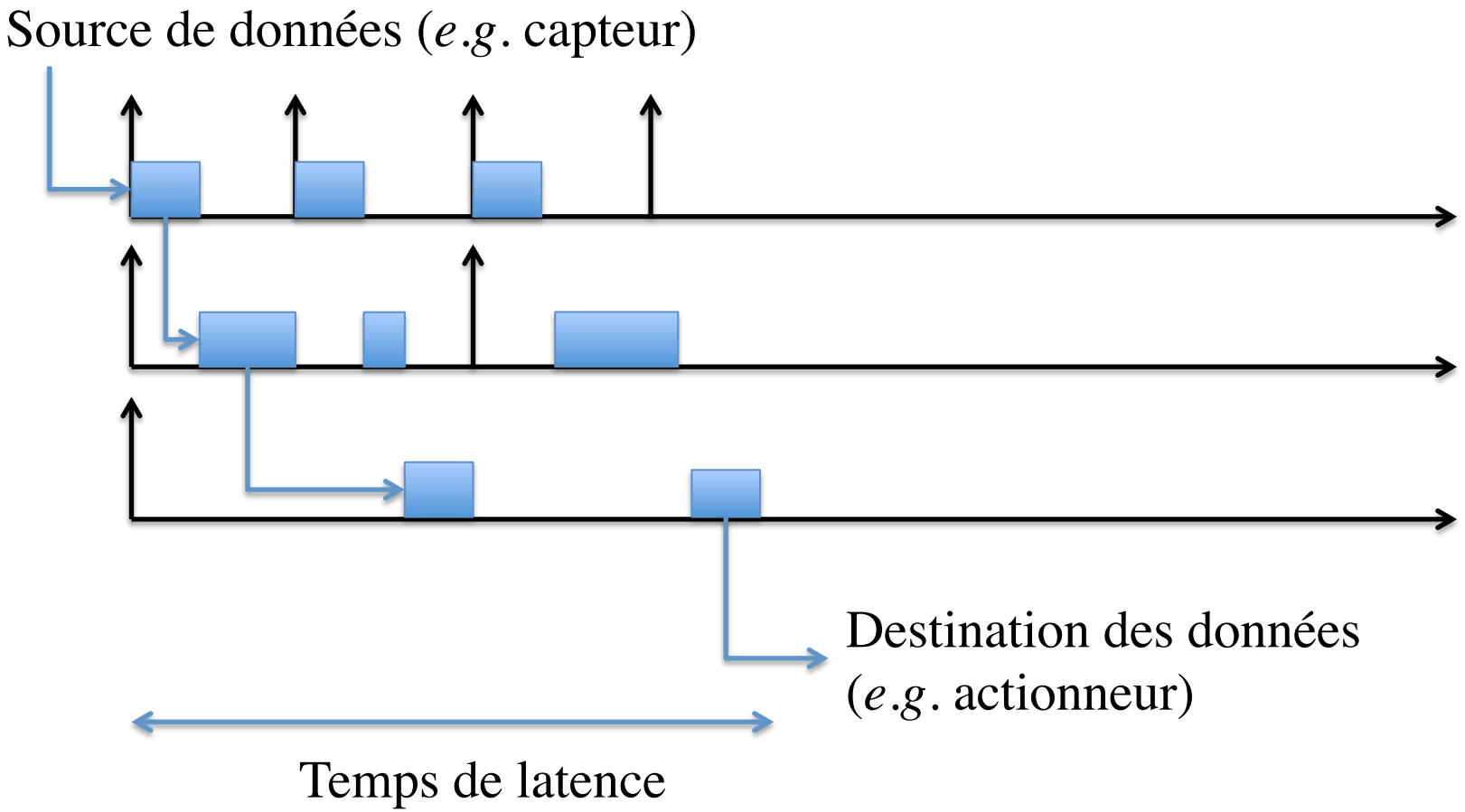

*Comment, et pourquoi modéliser une telle architecture?*


# Temps de latence et gigue, définition

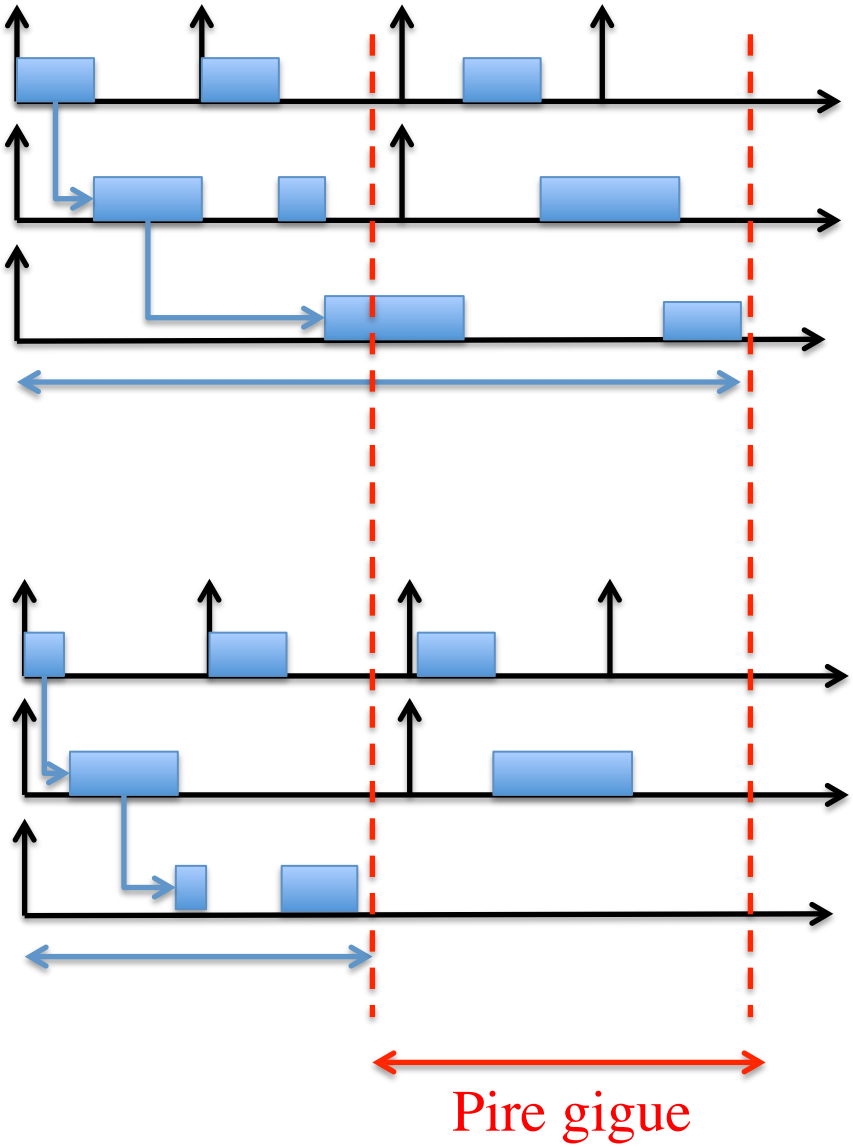
# Temps de latence, gigue: motivations

- Pourquoi s'intéresser au temps dans les systèmes embarqués?
  - Période d'une tâche = fréquence à laquelle une fonction du système est effectué; ex. influence importante sur la qualité du contrôle (stabilité, rapidité de convergence…)
  - Temps de latence = temps qui sépare la capture des données d'entrée de la production des données de sortie; ex. influence sur la sur la qualité du contrôle et sur la définition de la période d'échantillonnage
  - Gigue = variation du temps de latence, qui rend la production des données plus ou moins périodique.
- Le temps de latence et la gigue peuvent être considérées dans le pire cas, le meilleur cas, ou le cas moyen…
  - Dans ce cours, on s'intéressera principalement au pires cas
- Objectif: avoir un temps de latence et une gigue suffisamment petits

# Latence, illustration



# Gigue, illustration




Scénario 1:
pire temps de latence

Scénario 2:
meilleur temps de latence

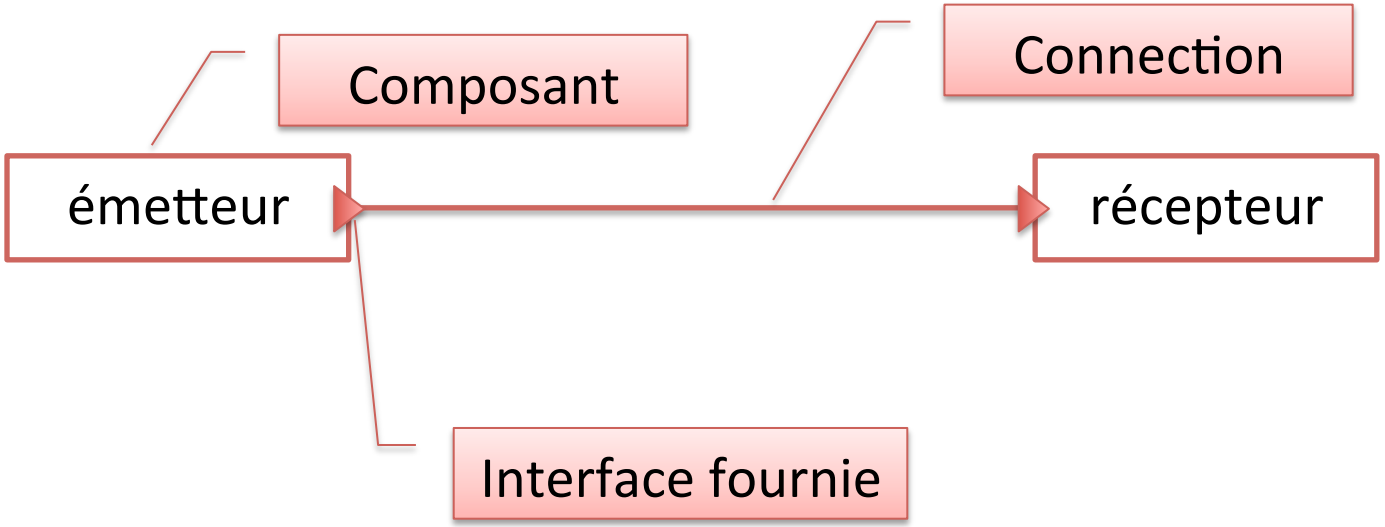
# Contributeurs au temps de latence

- Le temps de réponse, vu précédemment – Response Time Analysis, est un contributeur du temps de latence et de la gigue…
- Quels autres contributeurs? Les communications
  - ⌘ Entre tâches sur un même processeurs
  - ⌘ Entre tâches sur des processeurs différents
- En plus de l'exécution des tâches (vue en cours – Response Time Analysis) nous allons donc devoir nous intéresser aux communications entre tâches

- Il va donc falloir enrichir nos modèles avec des informations sur les communications entre composants…

# Les langages de description d'architecture (ADL)

- 3 éléments principaux :
  - les composants (ensemble d'interfaces requises et fournies)
  - les connections entre les composants (chemins d'échanges d'information)
  - une sémantique associée à ces éléments (comportement correspondant à une catégorie d'éléments)
- Les interfaces et connections entre composants définissent les chemins de communication (structure utile au calcul de temps de latence)




*Sémantique : émetteur transmet une donnée a récepteur*


# Les interfaces des composants AADL

# Les features— les ports

- Les ports modélisent les échanges d'information.
  - ⌘ data : transport de données ; comme dans un circuit électronique
  - ⌘ event : émission/réception d'un évènement
  - ⌘ event data : évènement + données ; comparable à un message

- les ports peuvent être déclarés en
  - ⌘ entrée (`in`)
  - ⌘ sortie (`out`)
  - ⌘ entrée-sortie (`in out`)

# Features: les accès aux composants

- Un composant peut indiquer qu'il requiert (`requires`) ou qu'il fournit (`provides`) un accès à un sous-composant
  - un bus, p.ex. pour un processor ou une memory
  - une data, p.ex. pour une donnée partagée entre plusieurs threads

  Representation graphique 

- Un composant thread ou data peut offrir des sous-programmes comme interface
  - un thread serveur
    - p.ex. dans le cas d'un appel de procédure distante (RPC)
  - un composant de donnée proposant des méthodes d'accès
    - analogie avec les classes des langages objets

  Representation graphique 

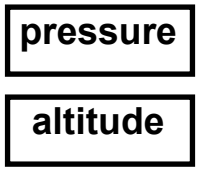
# Features: les groupes de ports

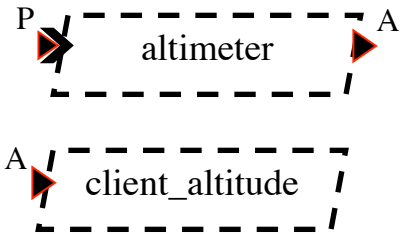
- Regroupement de ports associés entre eux
  - ⌘ facilite la manipulation au niveau de la description
  - ⌘ analogie avec un câble

# Exemple de features (composants logiciels)

```
data pressure
end pressure;

data altitude
end altitude;

-- Note: these are not
-- programming language data
-- types
```

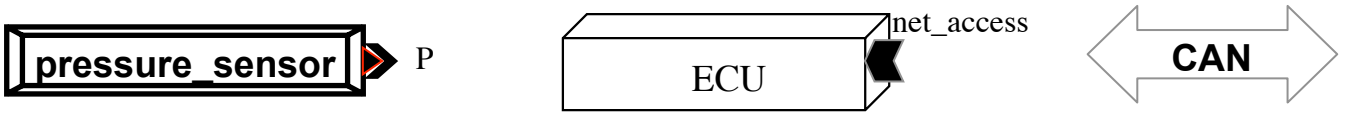
```
thread altimeter
features
  P : in event data port pressure;
  A : out data port altitude;
end altimeter;

thread client_altitude
features
  A : in data port altitude;
end client_altitude;
```



# Exemple de features (composants matériels)

```
device pressure_sensor
features
  P : out event data port pressure;
end pressure_sensor;

bus CAN
end CAN;

processor ECU
  features
    net_access: requires bus access CAN;
end ECU;
```

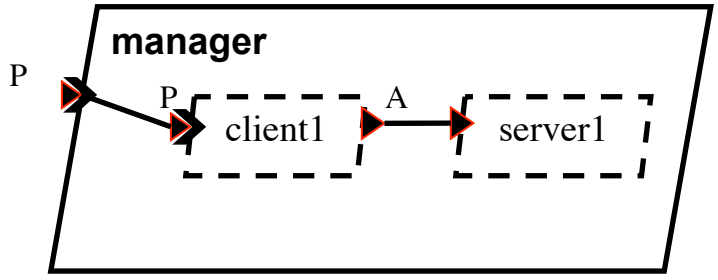


# Les connexions

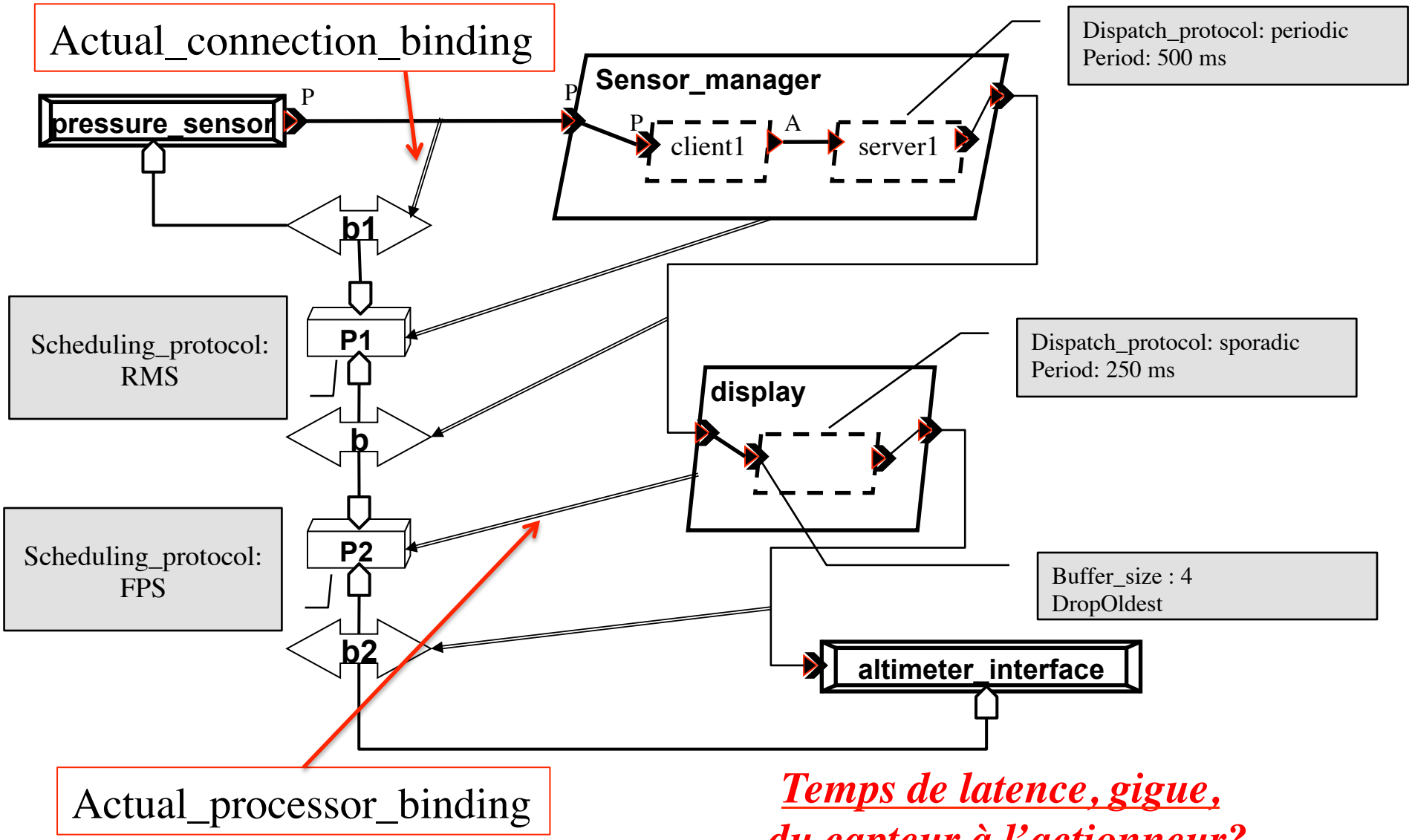
- Pour relier les « features » du même type entre elles
  - ports
  - Paramètres
  - accès aux sous-composants
  - …
- Les connexions sont définies dans les implementations de composants
  - Entre les interfaces du composant englobant et de ses sous-composants
  - Entre sous-composants inclus dans le même composant.
- Les features de sortie peuvent être connectés en « 1 vers n »
- `event data ports` & `event ports` entrants
  - Connections « n vers 1 » possible car gestion de files d'attente
- les in data port
  - Connections « n vers 1 » **impossibles**

# Exemple de connections

```
process manager
features
  P : in event data port pressure;
end manager;

process implementation manager.altitude
subcomponents
  client1 : thread client_altitude;
  server1 : thread altimeter;
connections
  pressure_input : port P -> client1.P;
  altitude_cnx: port client1.A -> server1.A;
end manager.altitude;
```



# Exemple complet: altimètre




***Temps de latence, gigue, du capteur à l'actionneur?***


# Les flux de données en AADL

# Les flux de données (1)

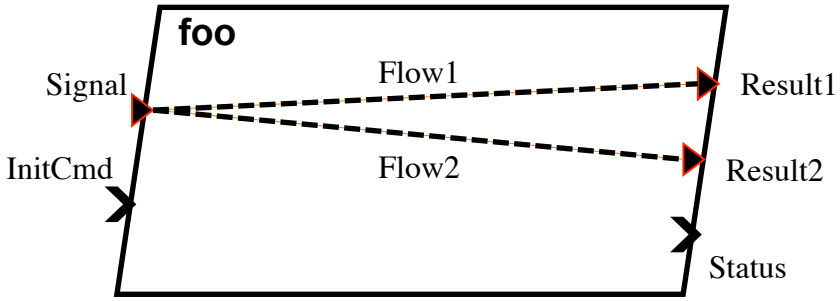
- Les flux permettent de matérialiser les chemins à travers les éléments d'une description.
  - Ils suivent plus ou moins les connexions, en tout cas s'appuient fortement dessus.
  - Représentation plus compacte pour pouvoir analyser plus facilement les flots de données et y associer des propriétés
- on peut décrire :
  - le début d'un flux (`flow source`)
  - le chemin d'un flux (`flow path`)
  - la fin d'un flux (`flow sink`)
  - un flux de bout en bout (`end to end flow`)

# Les flux de données (2)

- Les *component types* contiennent
  - des ***flow specifications*** (source, sink et path) qui ne font intervenir que des `features`
- les *component implementations* contiennent
  - des ***flow implementations*** (source, sink et path) qui font intervenir des features, des connections et des flots de sous-composants
  - des ***end to end flows*** qui commencent par un flow source et finissent par un flow sink.

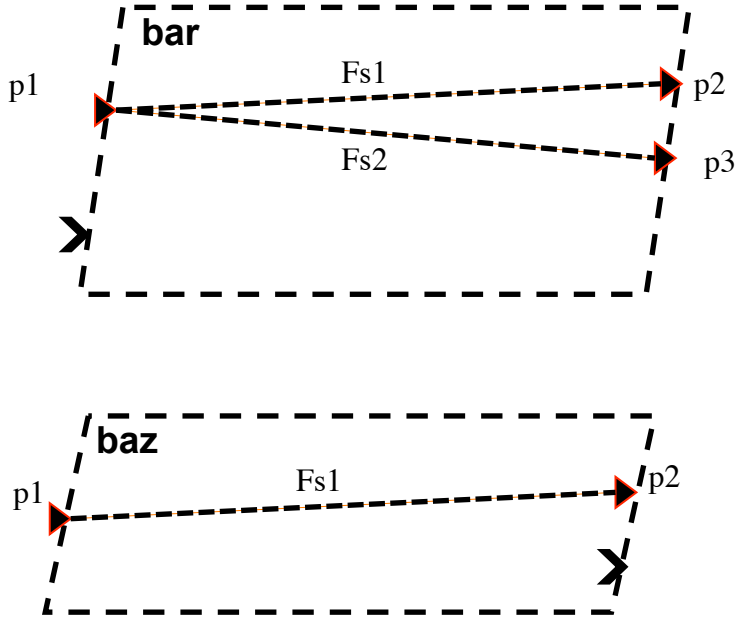
# Exemple de flow specification (1/2)

```
data position
end position;

data position_radial
end position_radial;

data position_cartesian
end position_cartesian;

process foo
features
  InitCmd : in  event port;
  Signal  : in  data port position;
  Result1 : out data port position_radial;
  Result2 : out data port position_cartesian;
  Status  : out event port;
flows
  Flow1 : flow path   Signal -> Result1;
  Flow2 : flow path   Signal -> Result2;
  Flow3 : flow source Status;
  Flow4 : flow sink   InitCmd;
end foo;
```



# Exemple de flow specification (2/2)

```
data angular_move
end angular_move;

thread bar
  features
    InitCmd : in  event port;
    p1 : in data port position;
    p2 : out data port angular_move;
    p3 : out data port position_cartesian;
  flows
    Fs1: flow path p1 -> p2;
    Fs2: flow path p1 -> p3;
    Fs3 : flow sink InitCmd;
end bar;

thread baz
  features
    p1: in data port angular_move;
    p2: out data port position_radial;
    Status  : out event port;
  flows
    Fs1: flow path p1 -> p2;
    Fs2 : flow source Status;
end baz;
```



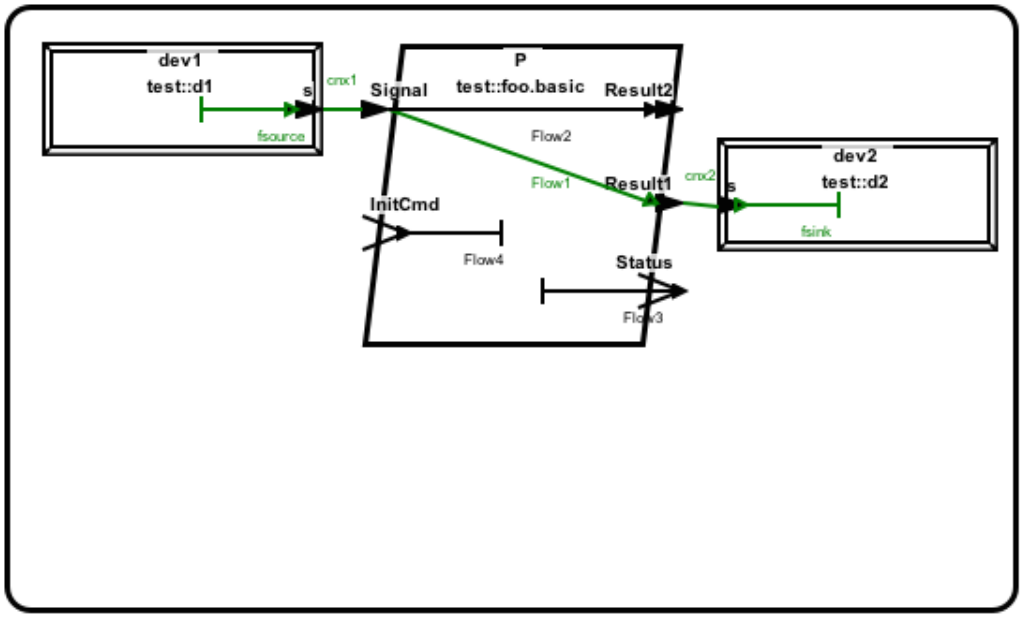
# Exemple de flow implementation (2/2)

```
process implementation foo.basic
  subcomponents
    A: thread bar;
    B: thread baz;
  connections
    Cnx1 : port signal   -> A.p1;
    Cnx2 : port A.p2     -> B.p1;
    Cnx3 : port B.p2     -> result1;
    Cnx4 : port A.p3     -> result2;
    Cnx5 : port B.status -> status;
    Cnx6 : port InitCmd -> A.InitCmd;
  flows
    Flow1: flow path signal -> Cnx1 -> A.fs1 -> Cnx2 -> B.fs1 -> Cnx3 -> result1;
    Flow2: flow path signal -> Cnx1 -> A.fs2 -> Cnx4 -> result2;
    Flow3: flow source B.fs2 -> Cnx5 -> status;
    Flow4: flow sink initcmd -> Cnx6 -> A.fs3;
end foo.basic;
```

Même nom que dans le flow specification (relation de « raffinement »), voir 2 slides plus haut

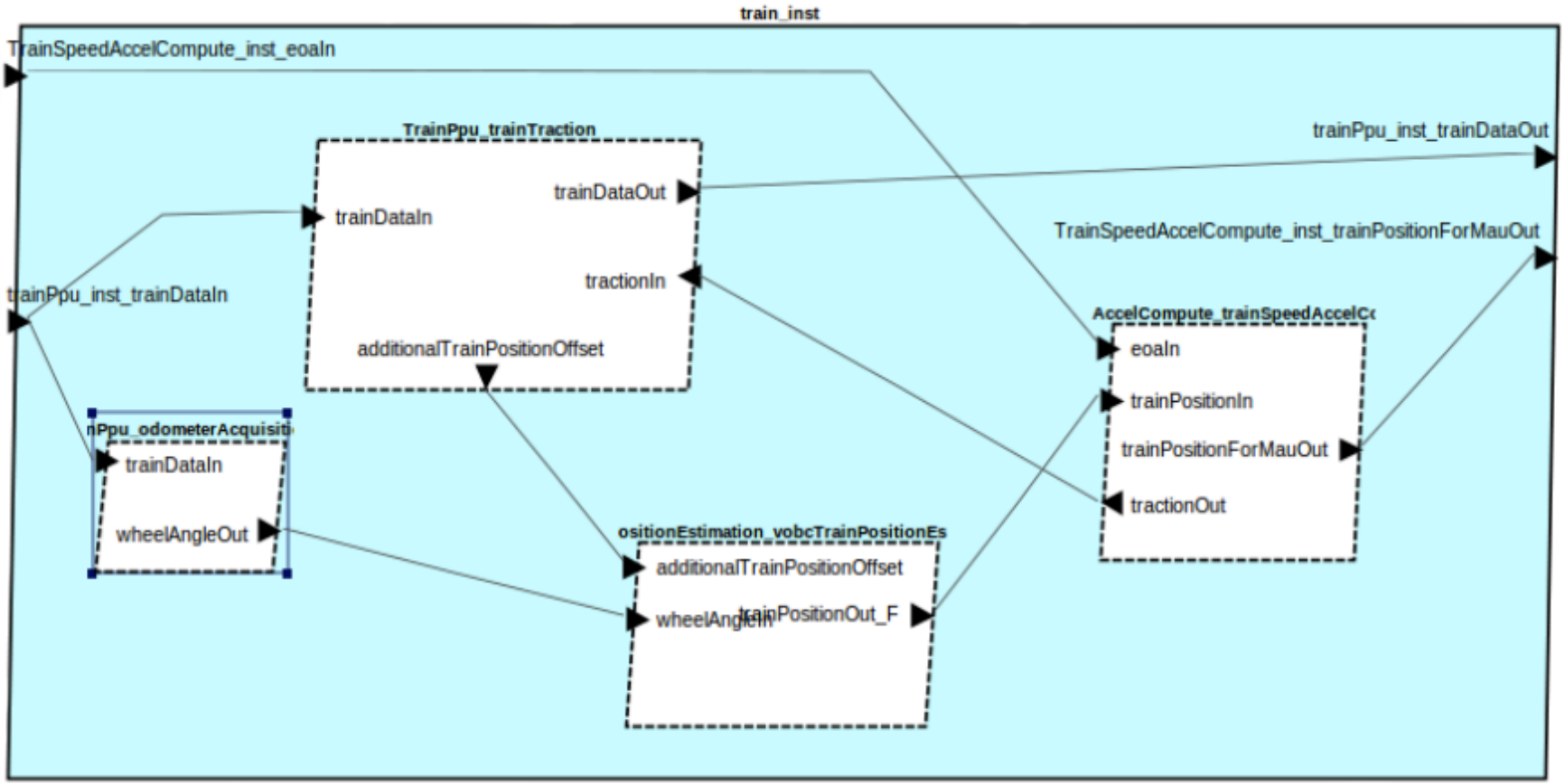
# Exemple de end to end flow

```
device d1
  features
    s: out data port Position;
  flows
    fsource: flow source s;
end d1;

device d2
  features
    s: in data port position_radial;
  flows
    fsink: flow sink s;
end d2;

system e2e_flow_spec
end e2e_flow_spec;

system implementation e2e_flow_spec.impl
  subcomponents
    P: process foo.basic;
    dev1: device d1;
    dev2: device d2;
  connections
    cnx1: port dev1.s -> P.Signal;
    cnx2: port P.Result1 -> dev2.s;
  flows
    e2e: end to end flow dev1.fsource -> cnx1 -> P.Flow1 -> cnx2 -> dev2.fsink;
  properties
    latency => 30 ms .. 50 ms applies to e2e;
end e2e_flow_spec.impl;
```

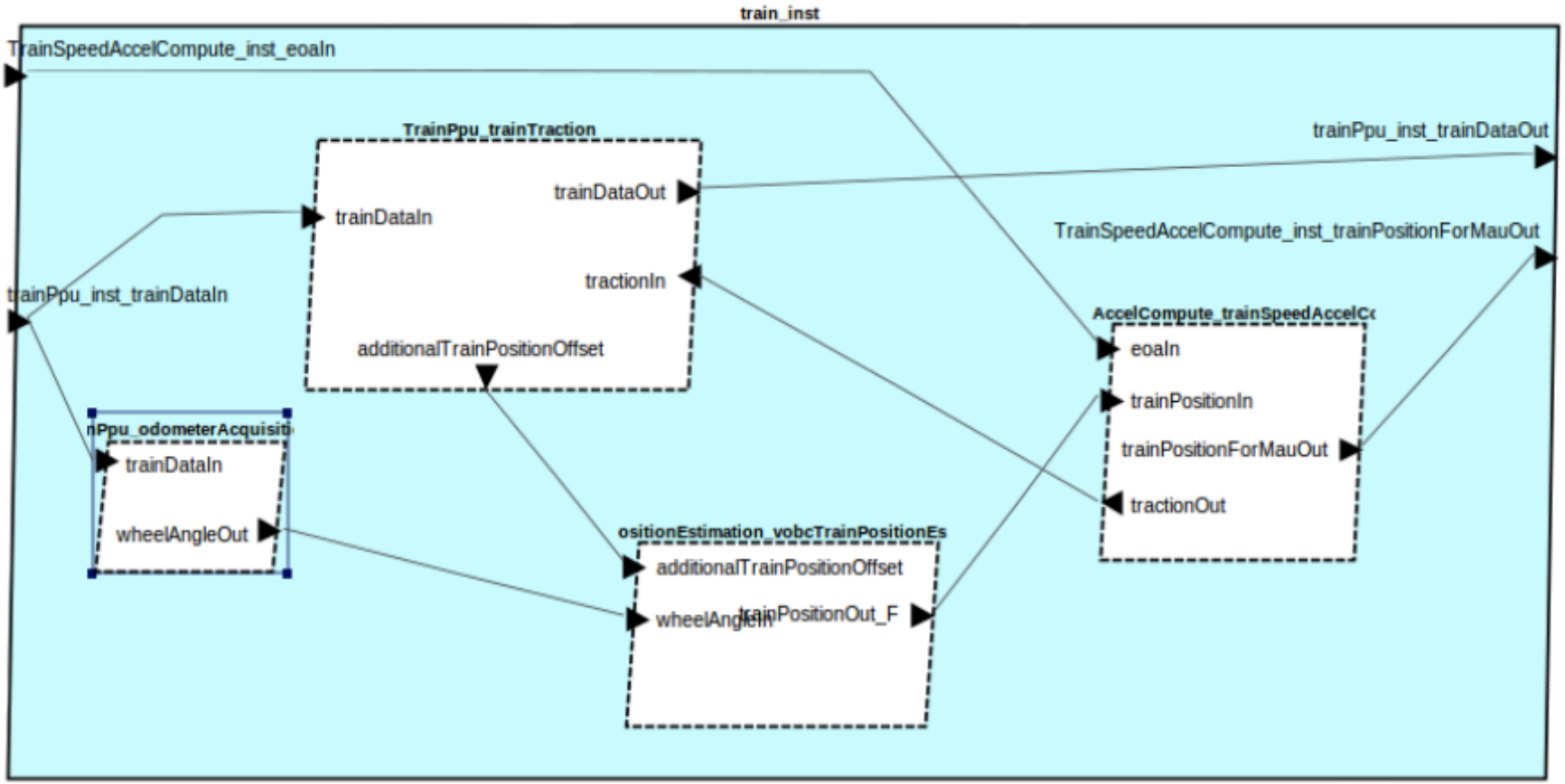



Spécification du temps de latence et de la gigue entre dev1 et dev2 (exigence)

# Redondant avec les connections?



# Redondant avec les connections?




**Les flows permettent de sélectionner des sous parties du système**

# Notes sur la modélisation présentée

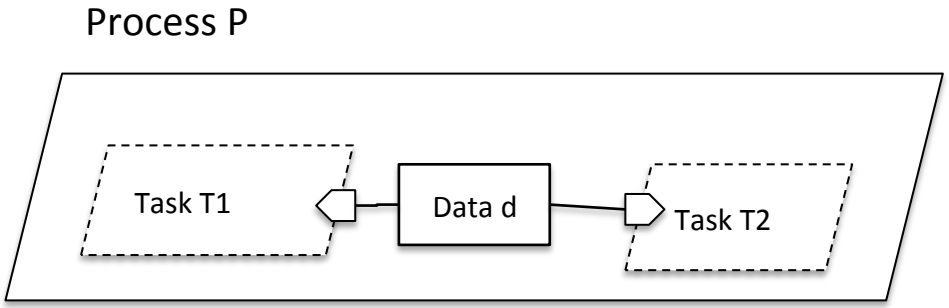
- Les flow permettent de sélectionner des sous parties du système
  - **Identification des sous parties d'intérêt**: toutes les fonctions d'un système ne seront pas modélisées avec le même niveau de détail ou de précision. Pour certaines, on se contentera de la définition des interfaces (e.g. pas temps réel dur ni critique). Pour d'autres il faudra aller jusqu'à l'estimation des WCET, WCRT, etc. (e.g. temps réel dur ou critique).
  - **Décomposition et séparation des rôles:** toute l'information associée à une architecture ne se trouve pas chez un unique acteur de la production d'un système (systémier/équipementier). Le modèle sert alors à intégrer les contributions des sous-traitants dans l'analyse d'une propriété plus globale, au niveau système par exemple.
  - **Imprécision:** Les informations présentes dans le modèle sont souvent incertaines (imprécises ou inconnues). Dans les premières phases de développement il s'agit de **budgets**. Ces budgets peuvent faire l'objet de discussions/négociations entre intégrateur et équipementier.


# Sémantique des liens de communications en AADL

# Différence de sémantique entre ports, et data access

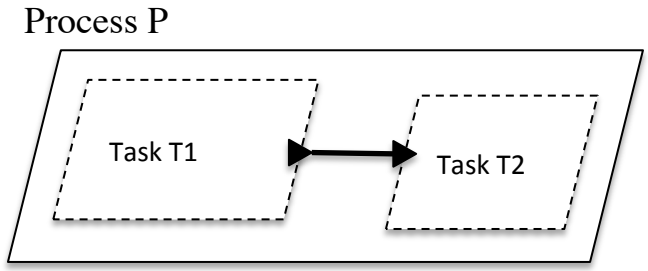
- Un **data port** représente une seule instance de la donnée échangée entre composants (pas de mise en file d'attente).
- Un **event** ou **event** data port représente une file de messages; Les messages non-utilisés pendant une activation d'une tâche sont perdus.
- Un data access représente l'accès à une donnée.
- La sémantique va être précisé via des propriétés AADL; sans modifier la valeur par défaut de ces propriétés, on a:
  - Data access: accès non-protégé en lecture/écriture avec lecture/écriture à n'importe quel moment de l'activation d'une tâche
  - Propriété Input_Time, pour tous les ports d'entrée (data/event/event data):
    - La donnée, l'événement, ou le message d'un port « in » sont lus en début d'activation du thread, et le thread travaillera avec cette copie tout au long de son activation sans qu'elle puisse être modifiée par l'écrivain ou le producteur
  - Propriété Output_Time, pour tous les ports de sortie(data/event/event data)
    - La donnée, l'événement, ou le message d'un port « out » sont écrit en fin d'exécution du thread

# Mode de communication simple: variables partagées entre threads




- Avantage: très simple à implémenter
- Inconvénient: peu d'informations exploitables pour l'analyse de temps de latence: qui écrit (T1 ou T2 ou les deux)? Qui lit (T1 ou T2 ou les deux), Quand? …

# Communication via des ports de données




- Les ports de donnée ont une **direction** (on sait quel tâche écrit, et/ou lit l'information), on sait aussi **quand** grâce aux propriétés suivantes
- Propriétés AADL qui influencent le calcul de temps de latence:
  - Deadline ou compute_execution_time
  - Input_Time/Output_Time (valeur par défaut décrites précédemment)
  - Timing
- Spécification du temps de latence (*i.e.* exigence) via la propriété
  - Latency

# Comment se fait le calcul de temps de latence…

- Contributeurs au temps de latence:
  1. Les temps de calcul (propriété **deadline** ou **compute_execution_time**).
  2. Le temps de communication entre processeurs (propriété ***Transmission_time***).
  3. Les délais de communication entre tâches sur un même processeur (propriété ***Timing***).

# Comment se fait le calcul de temps de latence…

1. Temps de calcul d'une tâche Ti est simple:
   - **Compute_Execution_Time** => X ms .. Y ms;
     Notez ici que Y correspond au WCRT (résultat de l'analyse RTA) et pas au WCET (donnée d'entré de l'analyse RTA…)
2. Le temps de communication entre processeur se modélise simplement en AADL:
   - Transmission_Time => [ Fixed => A ms .. B ms;
     PerByte => C ms .. D ms; ];
   - data_size => Z bytes;

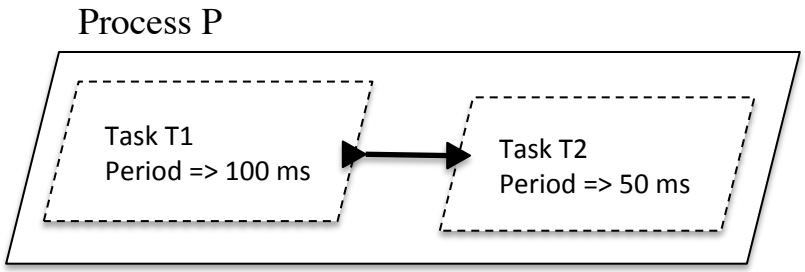
     Min(RemoteComTime) = A + C * Z

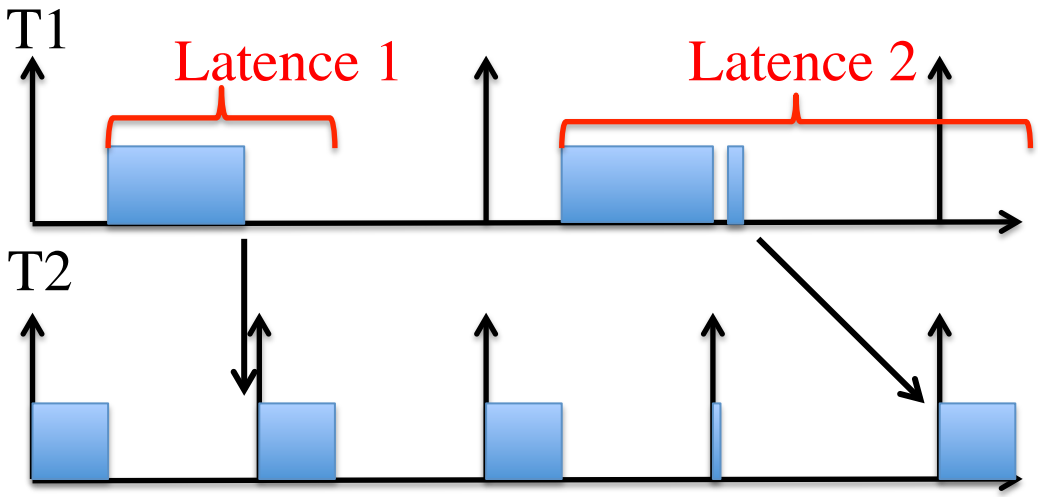
     Max(RemoteComTime) = B + D * Z

3. Reste le délais de communication entre tâches sur un même processeur…
   Propriété **Timing** prend pour valeur
   - sampled,
   - immediate,
   - delayed

# Communications *sampled* (valeur par défaut de la propriété Timing)

- Définition: le lien de communication est une simple variable partagée
  - en lecture (port in)/écriture (port out);
  - l'écrivain écrit à la fin de son exécution et le lecteur lit en début d'exécution)

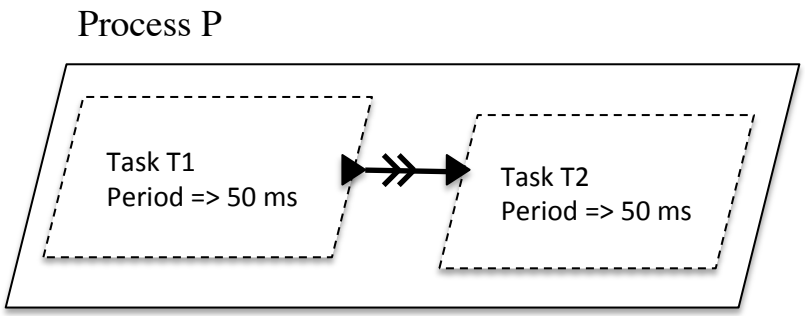


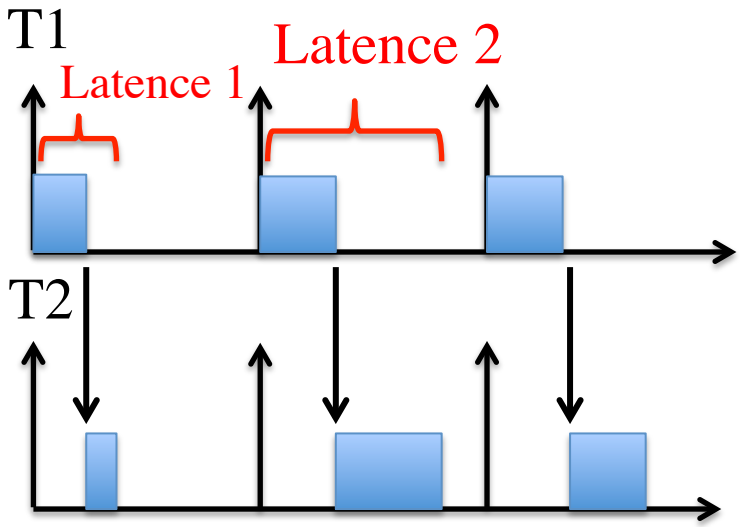




- Le traitement de la donnée se fait (calculs approximatifs)
  - au plus tôt après une exécution courte de T1 et une exécution courte de T2
  - au plus tard après une exécution longue de T1, une période de T2, et une exécution longue de T2

# Communications *immediate*

- Définition: une connexion *immediate* impose que le lecteur ne démarre qu'après avoir reçu une nouvelle donnée




- Le traitement de la donnée se fait
  - ⌘ au plus tôt après une exécution courte de T1 et une exécution courte de T2
  - ⌘ au plus tard après une exécution longue de T1 et une exécution longue de T2

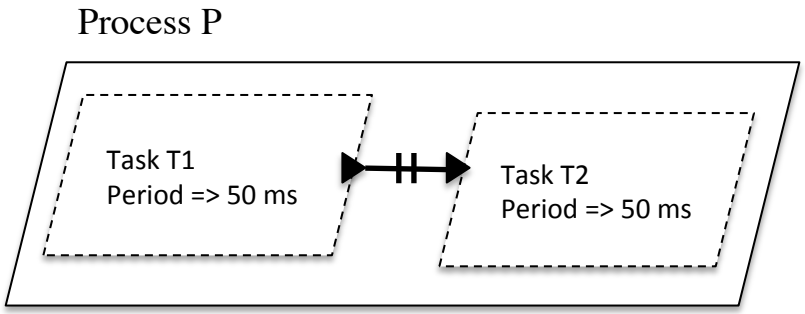
# Communications immediate

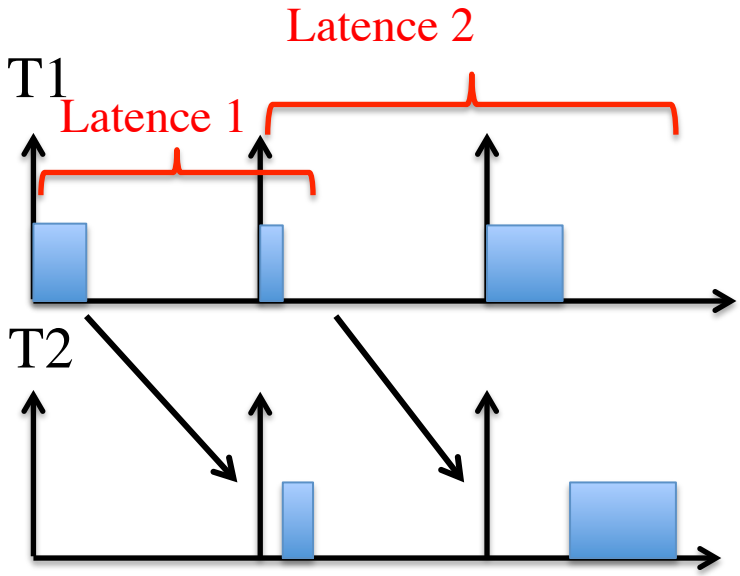
- Minimise le temps de latence
- Attention aux dépendances cycliques: T1 doit précéder T2 et T2 doit précéder T1… Il faut donc se limiter au DAGs
- Attention aux incohérences : T1 doit précéder T2 mais T2 est plus fréquente que T1… Il faut donc imposer des rapport de fréquence entre T1 et T2.

```
process implementation p.impl
   subcomponents
      t1_inst: thread t1;
      t2_inst: thread t2;
   connections
      cnx1: port t1_inst.p_out -> t2_inst.p_in;
   properties
      Timing => immediate applies to cnx1;
end p.impl;
```

# Communications *delayed*

- Définition: une connexion *immediate* impose que l'écrivain ne produise sa donnée qu'au moment de sa *deadline*




- Le traitement de la donnée se fait
  - ⌘ au plus tôt après période de T1 et une exécution courte de T2
  - ⌘ au plus tard après une période de T1 et une exécution longue de T2

# Communications *delayed*

- Minimise la gigue: différence entre BCRT et WCRT de la tâche réceptrice

```
process implementation p.impl
   subcomponents
      t1_inst: thread t1;
      t2_inst: thread t2;
   connections
      cnx1: port t1_inst.p_out -> t2_inst.p_in;
   properties
      Timing => delayed applies to cnx1;
end p.impl;
```

# Notes

- Dans les systèmes critiques ou temps-réel durs, on utilise le plus souvent possible les communications immediates, mais
  - ⌘ les cycles sont résolus en utilisant des communications delayed.
  - ⌘ Les problèmes de gigue (désynchronisation par exemple) sont résolus en utilisant des communications delayed.

- Les communications delayed et immediate peuvent-être implémentées sans verrou, c'est plus difficile à vérifier dans le cas de communications sampled

# Pour aller plus loin

- Syntaxe AADL complète:
  http://perso.telecom-paristech.fr/~borde/aadl/documents/AADL_V2_Syntax_Card.pdf
- Exemples de modèles AADL:
  https://github.com/OpenAADL/AADLib
  https://wiki.sei.cmu.edu/aadl/index.php/Models_examples
- Sites web AADL:
  http://www.aadl.info
  https://wiki.sei.cmu.edu/aadl/index.php/Main_Page

- Outils
  - http://penelope.enst.fr/aadl
  - https://wiki.sei.cmu.edu/aadl/index.php/Osate_2
  - http://www.ellidiss.com/products/aadl-inspector/
  - http://beru.univ-brest.fr/~singhoff/cheddar/index-fr.html