



# AADL: Architecture Analysis & Design Language

Etienne Borde

[etienne.borde@telecom-paristech.fr](mailto:etienne.borde@telecom-paristech.fr)



# Introduction et généralités

# Modèle = abstraction

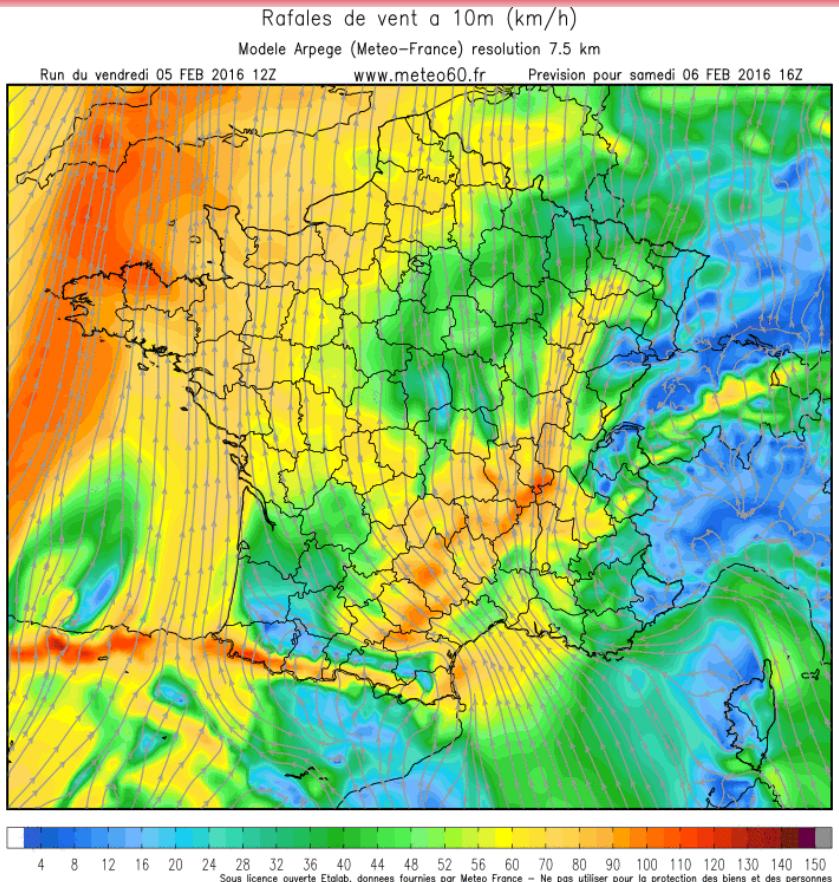
- Un modèle est une abstraction de la réalité, abstraction à partir de laquelle il devient possible de raisonner
  - ⌘ Calculer des **dimensions caractéristiques** (consommation énergétique, temps de réponse, température, etc.).
  - ⌘ Vérifier la **conformité** des dimensions caractéristiques vis-à-vis d'exigences (poids d'un avion, nombre de passager, besoin en carburant par km...).
  - ⌘ Fournir des **points de vus** différent en fonction des rôles, préoccupations et/ou expertises de chacun (constructeur ou utilisateur d'un train, d'un avion, d'une voiture).
- C'est une définition plutôt vague, et c'est le but: pour modéliser, il faut de la rigueur (objectif = calcul, vérification) et de l'ouverture d'esprit (moyen = abstraction, point de vu)...
- Prenons quelques exemple pour rendre tout cela plus concret...

# Définition de base

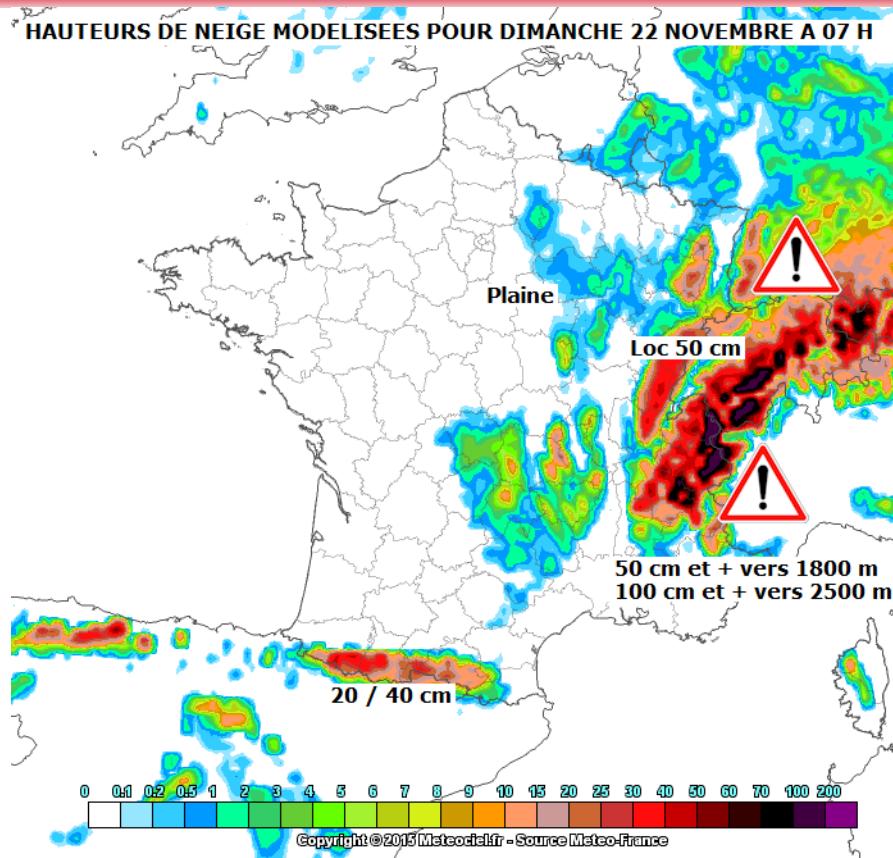
*Modeling, in the broadest sense, is the **cost-effective** use of something in place of something else for some **cognitive purpose**. It allows us to use something that is simpler, safer or cheaper than reality instead of reality for some purpose. A model represents reality for **the given purpose**; the model is an **abstraction of reality in the sense that it cannot represent all aspects of reality**. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality.*

*“The Nature of Modeling” Jeff Rothenberg in Artificial Intelligence, Simulation, and Modeling, L.E. William, K.A. Loparo, N.R. Nelson, eds. New York, John Wiley and Sons, Inc., 1989, pp. 75-92*

# Exemple ... Cartes météo



Carte des rafales de vents,  
avec direction et vitesse



Carte de l'enneigement, avec  
Risques d'avalanches

# Carte météo = abstraction

- Un modèle est une abstraction de la réalité

- ⌘ Calculer des **dimensions caractéristiques**

- *Niveau de risque d'inondation/incendie/avalanche*
    - *Température max/min/moyenne*
    - *Vitesse des rafales de vent*

- ⌘ Vérifier la conformité des dimensions caractéristiques vis-à-vis d'exigences

- *Un marin vérifiera la présence de vent, sa direction et sa vitesse...*
    - *Un alpiniste vérifiera la présence de neige, la présence de vent, la température...*

- ⌘ Fournir des **points de vus** différents en fonction des rôles, préoccupations et/ou expertises de chacun.

- *Carte des températures, des vents, des précipitations, etc...*

# Modèle = syntaxe + sémantique

- Un modèle a aussi pour objectif de donner une représentation facile à comprendre, interpréter, retenir...
- Il faut pour cela:
  - ⌘ Une représentation textuelle et/ou graphique.
  - ⌘ Une sémantique associée, plus ou moins abstraite.
- Sur les exemples précédents, les couleurs servent souvent de représentation graphique (syntaxique) alors que la légende explique la sémantique associée.
- La sémantique associée est plus ou moins abstraite...
- Les modèles sont souvent des moyens de « prédiction »...

# Les modèles en informatique

- Un programme écrit en C est-il un modèle? ...
  - ⌘ *cost-effective use of something in place of something else for some cognitive purpose.*
    - Le langage C a bien été conçu pour réduire le cout de développement/portage d'un système d'exploitation
    - C'est un langage de « haut niveau », objectif cognitif.
  - ⌘ *It allows us to use something that is simpler, safer or cheaper than reality instead of reality for some purpose. A model represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality*
    - Il est plus facile d'écrire du code en C qu'en assembleur, l'objectif est le même: automatiser l'exécution d'un algorithme sur un ordinateur
    - On s'abstrait du fonctionnement de la machine (jeu d'instruction du processeur, structure de la mémoire, registres, etc...).
- Est-ce suffisant? ... Analyse/Vérification?

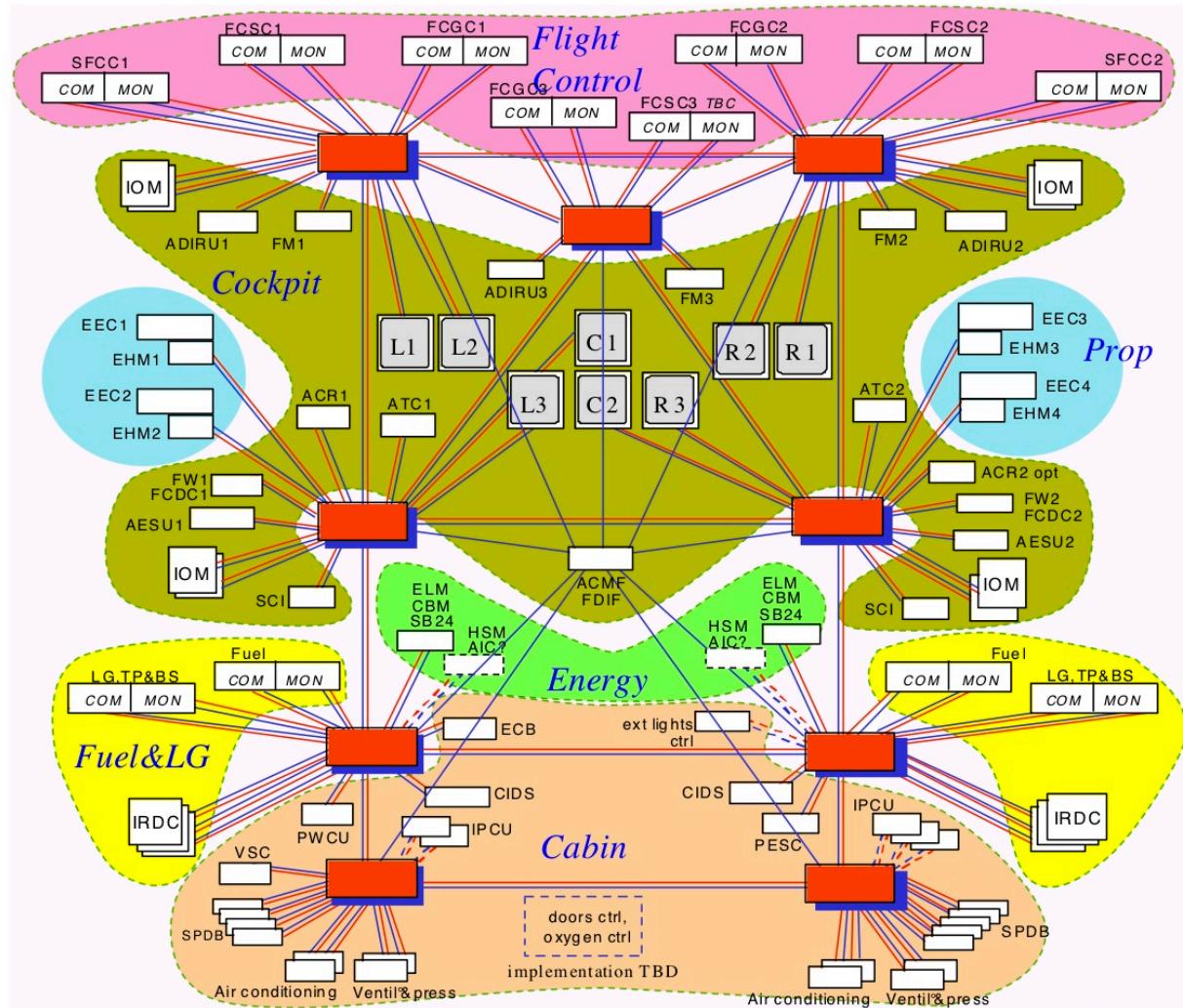
# Rôle des modèles pour les systèmes embarqués

- S'appuyer sur des modèles lors de la conception pour prendre des décisions, « valider » au plus tôt, explorer des alternatives ...
- Faciliter les phases de conception en s'appuyant sur une représentation abstraite (modèle):
  - ⌘ Support à la communication entre ingénieurs
  - ⌘ Support à la documentation des solutions choisi
  - ⌘ Support à la validation au plus tôt des exigences, par raisonnement sur le modèle (calcul de propriétés caractéristiques)
  - ⌘ Support à la configuration de la plate-forme (OS+routage)
  - ⌘ Support à la génération de code source (C, Ada, Java...)

# Modèles et architectures

- En informatique, les modèles permettent de représenter une **architecture** informatique: l'**organisation** et les **relations** existantes entre les **éléments** (logiciels, matériels, et/ou sous-systèmes) d'un système informatique.
- Rappel, un modèle utilise un langage :
  - ⌘ syntaxe + sémantique
- On parle de langage de description d'architecture

# Exemple d'architecture avionique



*Comment, et pourquoi modéliser une telle architecture?*

# Contexte industriel

- Systèmes de plus en plus complexe
  - ⌘ difficulté de compréhension du système
  - ⌘ difficulté de partager de l'information synthétique
  - ⌘ difficulté d'analyser le système
- Temps de commercialisation de plus en plus court
  - ⌘ Automatiser le processus de construction
  - ⌘ Réutiliser et adapter des fonctionnalités existantes
- Un formalisme de description est nécessaire
  - ⌘ pour décrire l'architecture (documentation, partage de l'information au sein d'une équipe technique)
  - ⌘ pour analyser et vérifier ses propriétés
  - ⌘ pour générer le code (ou les squelettes de code) correspondant à cette architecture

# 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
  - ⌘ representation 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
- Langage trop complexes → apprentissage long et couteux
- Objectifs:
  - ⌘ Spécification,
  - ⌘ Documentation,
  - ⌘ Conception,
  - ⌘ Analyse,
  - ⌘ Génération de code



# Les composants AADL, et leur composition

# Caractéristiques générales

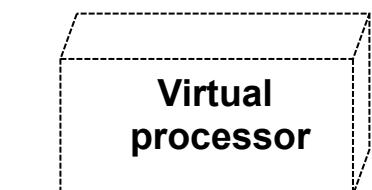
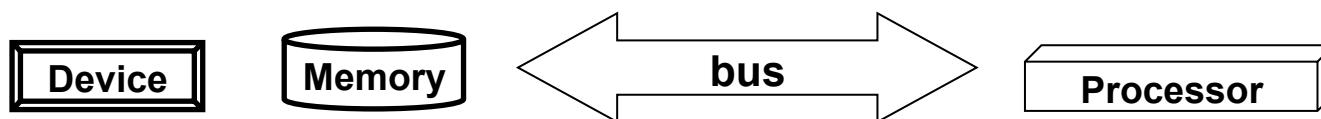
- Langages de description d'architecture: les **composants** sont les éléments de base du langage.
- En AADL, les composants appartiennent à une Catégorie (Process, thread, data, processor, etc.)
- La déclaration d'un composant est structurée en
  - ⌘ Déclaration du type de composant: vise à définir ses interfaces de communications avec d'autres composants (ports, access points, ...)
  - ⌘ Déclaration de l'implémentation de composant: vise à définir la structure interne du composant (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 sensif 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
  - ⌘ Virtual processor: resource virtuelle d'exécution (e.g. machine virtuelle)
  - ⌘ memory : composant de stockage (disque dur, mémoire vive, 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

System

# 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.

<u>Implémentation de catégorie</u>	<u>Peut contenir des sous-composant de catégorie</u>
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 named Data_Model (if any),  
-- or to a property set named 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 nxtImpl
end nxtImpl; -- declaration of component implementation nxtImpl,
               -- of type nxt (declared above), of category system

processor arm
end arm;

processor implementation armv7
end armv7; -- declaration of component implementation armv7,
               -- 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 nxtImpl
    subcomponents -- subcomponents section in nxtImpl
        cpu1: processor arm.v7;
        cpu2: processor arm.v7;
        proc1: process proc.impl;
        proc2: process proc.impl;
    end nxtImpl;

-- 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;
```

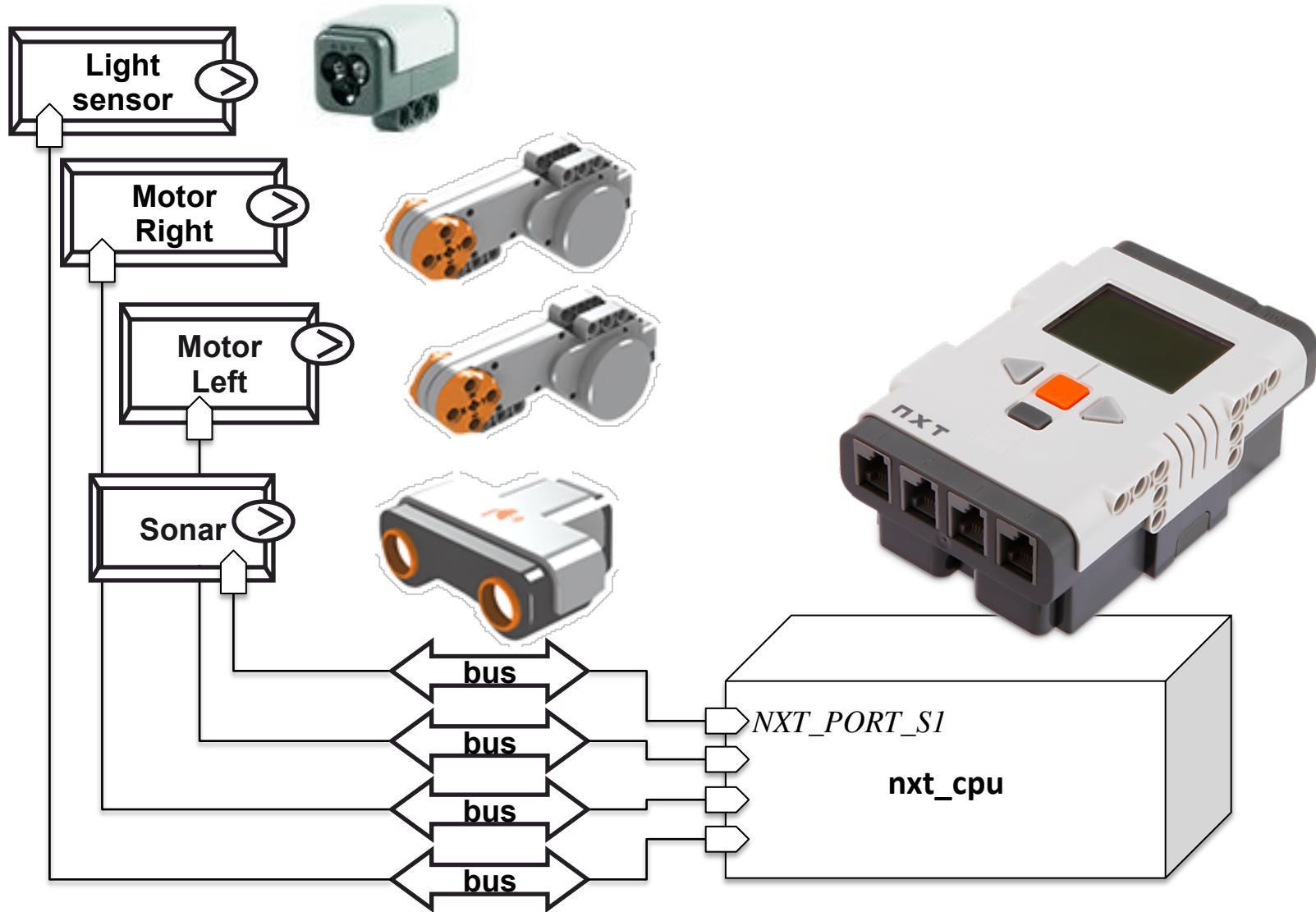
- Déclaraction d'un type de composant cpt\_id

```
process cpt_id
end cpt_id;
```

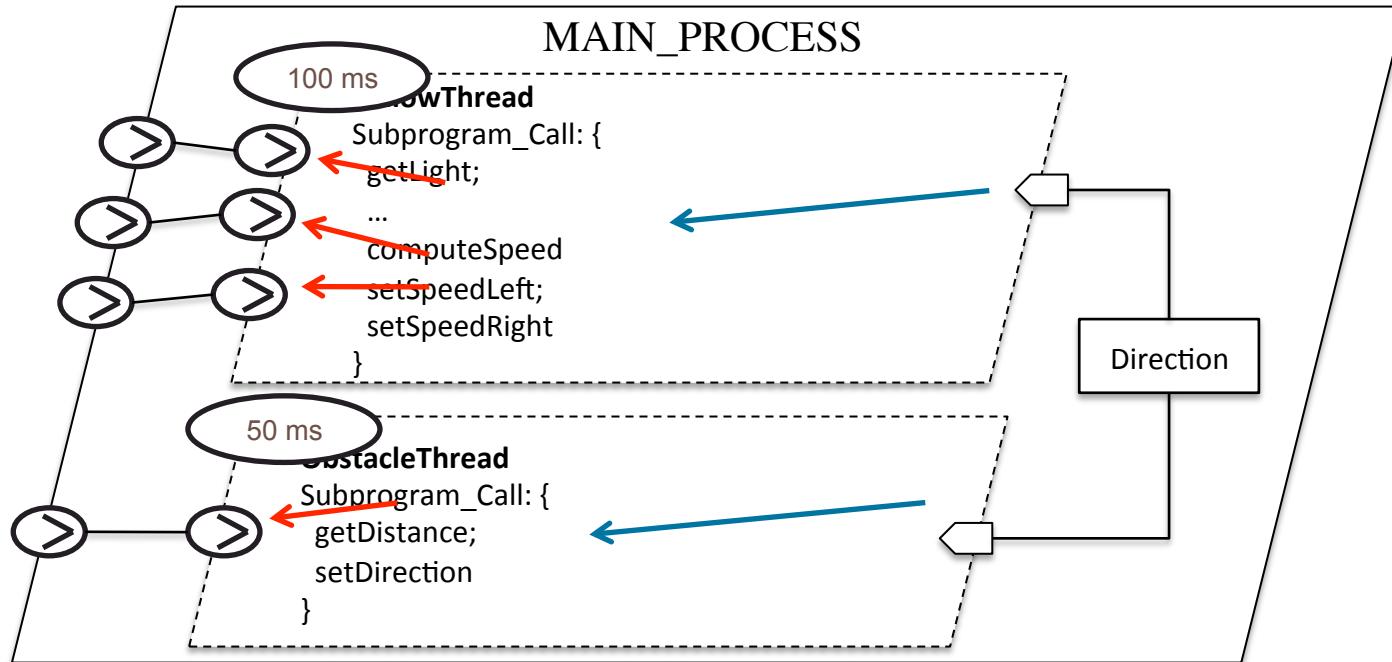
- Déclaraction d'une implémentation de composant cpt\_id.impl\_id, avec un sous composant subcpt\_id.

```
process implementation cpt_id.impl_id
subcomponents
    subcpt_id: thread other_cpt_id[.other_impl_id];
end cpt_id.impl_id;
```

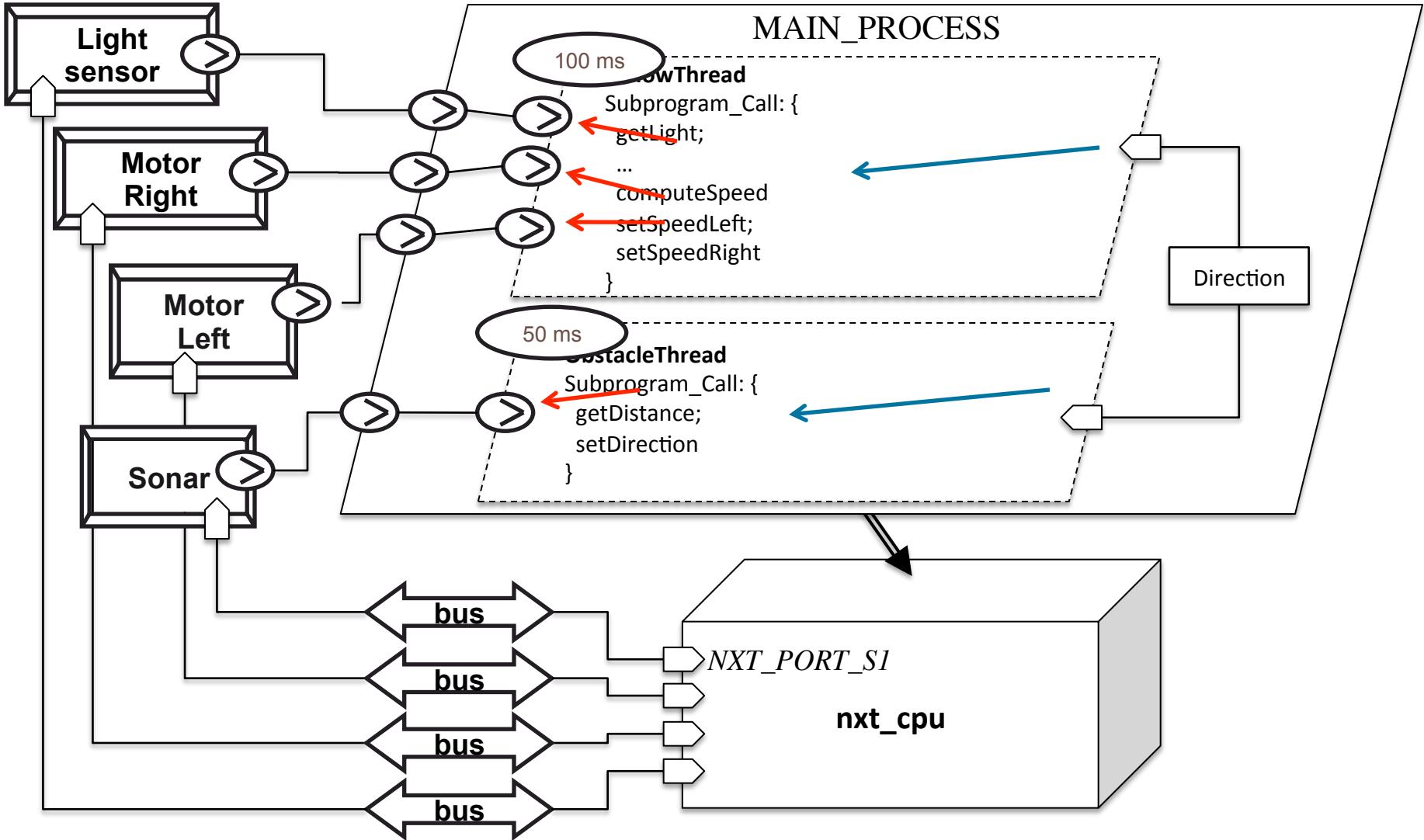
# Représentation des composants matériels du robot en AADL



# Intégration du code dans une architecture temps-réel



# Représentation de l'architecture intégrée du robot





## 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 élément : **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 pour l'analyse d'ordonancement

# 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 délai minimal 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** Identifiant de la propriété ou du type

**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;**

# 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**

Il s'agit d'un type de valeur

Time: **type adlinteger 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;**

# 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
```

Les valeurs admises pour ce type sont  
des entiers entre 0 et Max\_Time

```
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;
```

# 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
```

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

```
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;
```

# 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;

Time\_Range est un type admettant comme valeur des intervalles de temps

```
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;
```

# 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\_Proocols

**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_Proocols
        applies to (virtual processor, processor, system);
```

```
end Deployment_Properties;
```

NB: ce langage peut être utilisé pour définir des propriétés non-standardisée

# 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;
```

→ Politique d'ordo = ?

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

```
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;
```

...

# Exemples d'associations de propriété (scheduling protocol)

```

package nxt_use_case
public

system nxt end nxt;

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

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)!

# 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

AADL inspector (/home/borde/Install/AI-1.5-beta-patched/examples/patterns/synchronous.aadl)

File View Wizards Tools ?

Behavior\_Properties Data\_Model Base\_Types HW synchronous > Static Analysis Schedulability AI Scripts

```

346 SUBCOMPONENTS
347   T1 : THREAD a_thread
348     { Dispatch_Protocol => Periodic;
349       Compute_Execution_Time => 5 ms .. 5 ms;
350       Period => 15 ms;
351       Deadline => 15 ms; };
352   T2 : THREAD a_thread
353     { Dispatch_Protocol => Periodic;
354       Compute_Execution_Time => 5 ms .. 5 ms;
355       Period => 20 ms;
356       Deadline => 20 ms; };
357   T3 : THREAD a_thread
358     { Dispatch_Protocol => Periodic;
359       Compute_Execution_Time => 5 ms .. 5 ms;
360       Period => 25 ms;
361       Deadline => 25 ms; };
362 CONNECTIONS
363   C0 : PORT input -> T1.input;
364   C1 : PORT T1.output -> T2.input;
365   C2 : PORT T2.output -> T3.input;
366   C3 : PORT T3.output -> output;
367 END my_process.others;
368
369 THREAD a_thread
370 FEATURES

```

THE SIM

my\_process

T1 T2 T3

11

0 5 10 15 20 25 30 35 40 45 50 55 60 65 70

my\_process

T1 T2 T3

11

0 5 10 15 20 25 30 35 40 45 50 55 60 65 70

Static Analysis Schedulability AI Scripts

THE SIM

test	entity	
<input checked="" type="checkbox"/> processor utilization factor	root.my_platform.CPU	We can not prove that the task
<input checked="" type="checkbox"/> worst case task response time	root.my_platform.CPU	All task deadlines will be met :
response time	root.my_platform.CPU.my_process.T	15.00000
response time	root.my_platform.CPU.my_process.T	10.00000
response time	root.my_platform.CPU.my_process.T	5.00000

Static Analysis Schedulability AI Scripts

THE SIM

test	entity	
<input checked="" type="checkbox"/> Task response time computed from simulation	root.my_platform.CPU	No deadline missed in the computed scheduling : the ta
Number of preemptions	root.my_platform.CPU	0
Number of context switches	root.my_platform.CPU	46
Task response time computed from simulation	root.my_platform.CPU.my_process.T	worst = 5, best = 5 and average = 5.00000
Task response time computed from simulation	root.my_platform.CPU.my_process.T	worst = 10, best = 5 and average = 6.66667
Task response time computed from simulation	root.my_platform.CPU.my_process.T	worst = 15, best = 5 and average = 9.16667

# Modélisation des données partagées

- **Modélisation structurelle**

- ⌘ Variable globale: sous-composant de catégorie *data* dans une implémentation de composant *process*.
- ⌘ Accès aux données au niveau des threads: *feature* (i.e. interface) de type *requires data access* sur les thread
- ⌘ Connections entre les interface des threads et la variable partagée
- ⌘ Propriétés pour spécifier le mécanisme de protection

- **Modélisation comportementale**

- ⌘ Temps de calcul, prise et relâche de verrous

# Exemple (1/2)

Type de donnée (pas au sens langage de programmation)

```
DATA pressure
END pressure;
```

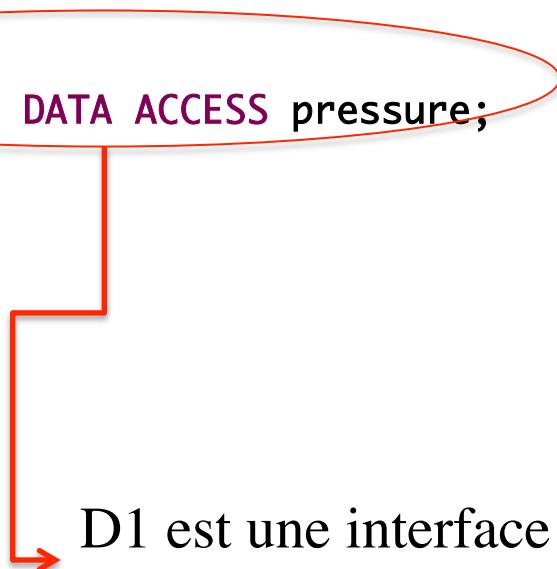
```
THREAD T
FEATURES
  D1 : REQUIRES DATA ACCESS pressure;
END T;
```

```
THREAD IMPLEMENTATION T.i1
PROPERTIES
  Dispatch_Protocol => Periodic;
  Compute_Execution_Time => 8ms..8ms;
  Period => 15 ms;
ANNEX Behavior_Specification {**
  states
    s : initial complete final state;
    transitions
      t : s -[on dispatch]-> s {
        D1 !<;
        computation(2ms);
        D1 !>
      };
    **};
END T.i1;
```

# Exemple (1/2)

```
DATA pressure
END pressure;
```

```
THREAD T
FEATURES
  D1 : REQUIRES DATA ACCESS pressure;
END T;
```



D1 est une interface  
d'accès à une donnée de type pressure

## THREAD IMPLEMENTATION T.i1 PROPERTIES

```
Dispatch_Protocol => Periodic;
Compute_Execution_Time => 8ms..8ms;
Period => 15 ms;
ANNEX Behavior_Specification {**
  states
    s : initial complete final state;
    transitions
      t : s -[on dispatch]-> s {
        D1 !<;
        computation(2ms);
        D1 !>
      };
    **};
END T.i1;
```

# Exemple (1/2)

```
DATA pressure
END pressure;
```

```
THREAD T
FEATURES
  D1 : REQUIRES DATA ACCESS pressure;
END T;
```

## THREAD IMPLEMENTATION T.i1 PROPERTIES

```
Dispatch_Protocol => Periodic;
Compute_Execution_Time => 8ms..8ms;
Period => 15 ms;
```

### ~~ANNEX Behavior\_Specification {\*\* states~~

```
s : initial complete final state;  
transitions
  t : s -[on dispatch]-> s {
    D1 !<;
    computation(2ms);
    D1 !>
  };
**};
```

```
END T.i1;
```

Comportement du thread

# Exemple (2/2)

**PROCESS IMPLEMENTATION** my\_process.others

**SUBCOMPONENTS**

```

T1 : THREAD T.i1;
D : DATA pressure {
    Concurrency_Control_Protocol => Priority_Ceiling;
};

T2 : THREAD T.i2;

```

**CONNECTIONS**

```

C1 : DATA ACCESS D -> T1.D1;
C2 : DATA ACCESS D -> T2.D1;

```

**END** my\_process.others;

Variable globale

Accès à la variable globale

# Exemple (2/2)

**PROCESS IMPLEMENTATION** my\_process.others

## SUBCOMPONENTS

T1 : THREAD T.i1;

D : DATA pressure {

Concurrency\_Control\_Protocol => Priority\_Ceiling;

} ;

T2 : THREAD T.i2;

## CONNECTIONS

C1 : DATA ACCESS D -> T1.D1;

C2 : DATA ACCESS D -> T2.D1;

**END** my\_process.others;

Mécanisme de protection





Plus généralement: les features (i.e.  
interfaces) des composants AADL

# Les features— les ports

- **Les ports modélisent les échanges d'information.**
  - ▶ ≈ data : transport de données ; comme dans un circuit électrique
  - ▶ ≈ 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 - 1

- 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 ▶

# Exemple de features

```
data pressure
end pressure;
```

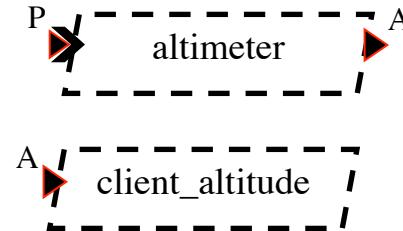
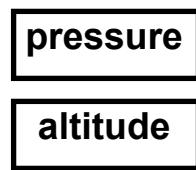
```
data altitude
end altitude;
```

```
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;

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

subprogram compute_altitude
features
  P : in parameter pressure;
  A : out parameter altitude;
end compute_altitude;
```



# Différence de sémantique entre ports, et data access

- La sémantique va être précisé via des propriétés;
- Par défaut:
  - ⌘ Tous les ports (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
    - La donnée, l'événement, ou le message d'un port « out » sont écrit en fin d'activation du thread
  - ⌘ Un data port
    - pas de file d'attente
  - ⌘ Un event ou event data port:
    - File d'attente
    - Les message non-utilisé pendant une activation sont perdus en fin d'activation
  - ⌘ Un data access:
    - Accès en lecture/écriture avec lecture à n'importe quel moment de l'activation
    - Accès non-protégé



## Les connexions des composants AADL

# Les connexions

- Pour relier les « features » entre elles
  - ⌘ ports
  - ⌘ paramètres
  - ⌘ sous-programmes d'interface
  - ⌘ accès aux sous-composants
  - ⌘ groupes de ports
- Les connexions sont définies dans les implementations de composants
- Les features de sortie peuvent être « 1 vers n »
- event data ports & event ports entrants
  - ⌘ « n vers 1 » car gestion de files d'attente
- les autres features entrantes
  - ⌘ « 1 vers 1 »

# Exemple de connexion

```

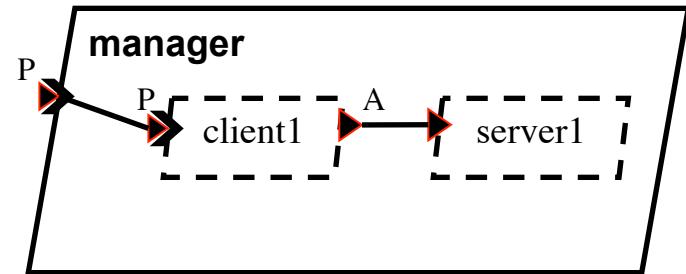
process manager
features
  P : in event data port pressure;
end manager;

thread implementation altimeter.basic
calls {
  appli : subprogram compute_altitude;
};

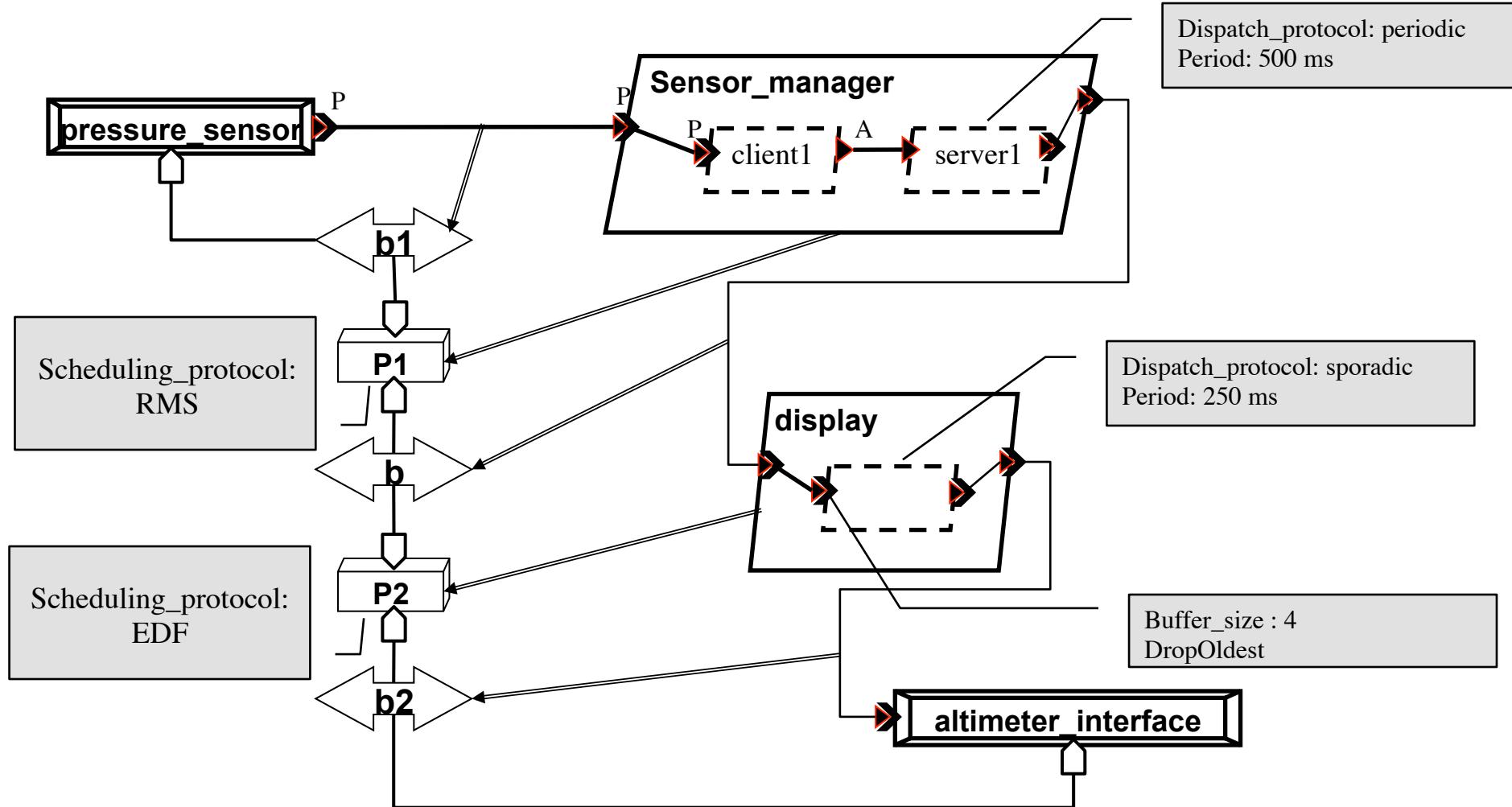
connections
  parameter P -> appli.P;
  parameter appli.A -> A;
end altimeter.basic;

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

```



# Exemple complet: altimètre





## Structuration d'une modélisation AADL

# Instanciation de l'architecture

- Une description AADL est une suite de déclarations
- Pour exploiter un modèle AADL, il est nécessaire de disposer de son modèle d'instance
  - ⌘ arborescence d'instances de composants AADL correspond aux déclarations de sous-composants
    - Valeurs de propriétés résolues
  - ⌘ filtre les éléments de la description qui ne sont pas référencés à partir du système racine

# Example de modèle d'instance en AADL

- Une modélisation AADL est un ensemble de déclarations de composants
- Les sous-composants sont des instances des déclarations
- Un système doit jouer le rôle de racine pour l'architecture
  - ⌘ pas d'interface
  - ⌘ une implémentation contenant les sous-composants

```
system global_system
end global_system;
```

```
system implementation global_system.two_machines
Subcomponents
  machine_1 : system machines::a_machine.mono_processor;
  machine_2 : system machines::a_machine;
end global_system.two_machines;
```

Instances

# Organisation d'une description

- Les paquetages (packages)
  - ⌘ matérialisent des espaces de nom
  - ⌘ une partie publique : visible de partout
  - ⌘ une partie privée : uniquement visible depuis le paquetage
- Les systèmes permettent de structurer l'architecture
- Les paquetages permettent de structurer la description

# Exemple d'utilisation des paquetages

```

package machines
public
  system a_machine
  end a_machine;

  system implementation a_machine.mono_processor
  subcomponents
    the_processor : processor machines::elements::a_processor.specific;
  end a_machine.mono_processor;
private
  system a_private_machine
  end a_private_machine;
end machines;

package machines::elements
public
  processor a_processor
  end a_processor;

  processor implementation a_processor.specific
  end a_processor.specific;
end machines::elements;

```



# Modélisation du comportement des threads et subprograms en AADL

# Appels de sous-programmes

- Une implantation de sous-programme ou de thread peut contenir des séquences d'appels à des sous-programmes
- L'ordre des appels est important
- Premier niveau de description du flux d'exécution dans les composants

```
subprogram spg1 end spg1;
subprogram spg2 end spg2;
thread a_thread end a_thread;

thread implementation a_thread.example
calls {
    call1 : subprogram spg1;
    call2 : subprogram spg2;
    call3 : subprogram spg1;
};
end a_thread.example;
```

# Représenter un passage de paramètres

```

data a_data
end a_data;

subprogram prog1
features
  input : in parameter a_data;
  output : out parameter a_data;
end prog1;

subprogram prog2
features
  input : in parameter a_data;
  output : out parameter a_data;
end prog2;

```

```

subprogram implementation prog2.impl
calls {
  a_call : subprogram prog1;
};

connections
  parameter input -> a_call.input;
  parameter a_call.output -> output;
end prog2.impl;

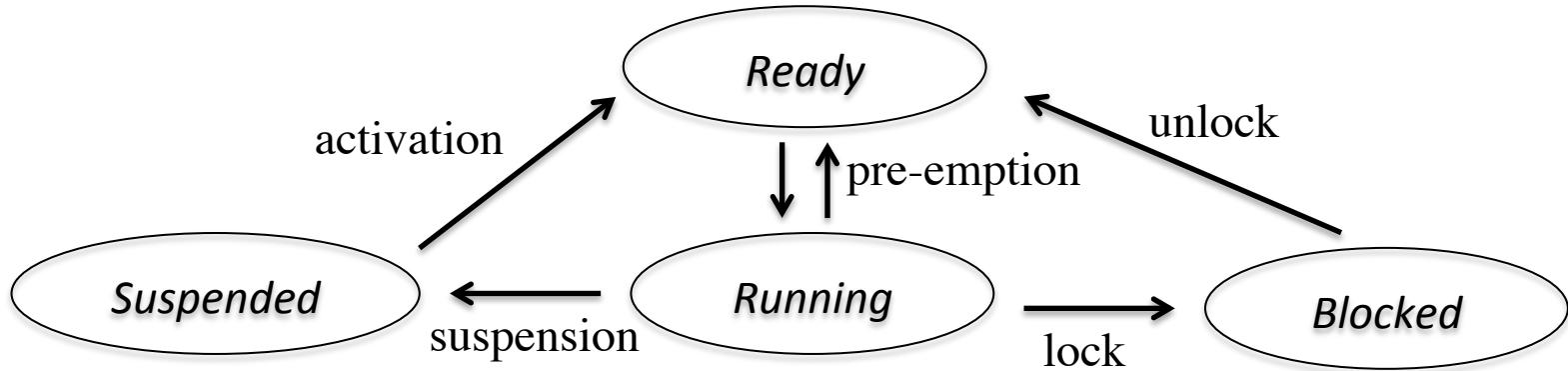
```

# Annexe comportementale

- Annexe standardisée
- Modéliser le comportement logiciel (thread, sous-programmes) par le biais de machines à état
- Une clause comportementale AADL est composée de trois parties:
  - ⌘ Déclaration des états
  - ⌘ Déclaration des transition entre ces états
  - ⌘ Déclaration des variables

# Les états d'une tâches

- Dans une analyse d'ordonnancement, on considère un système dans lequel un seul processus est constitué de N tâches (ou *threads*)
- Une tâche a quatre états:

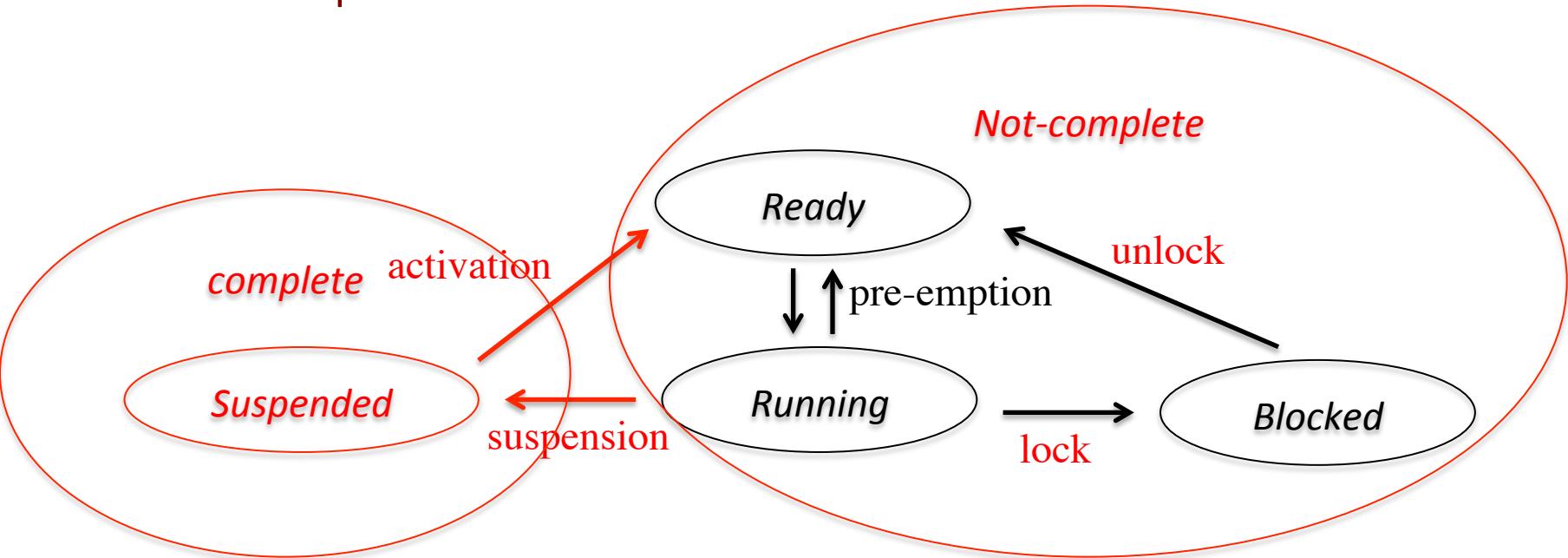


Remarques:

- les état « Bloqué » et « Suspendue » sont très similaires, et auraient pu être confondus. On distingue cependant ces deux états pour identifier
  - ⌘ le cas où la tâche a fini son exécution et attend la prochaine activation (*suspended*)
  - ⌘ le cas où la tâche est bloquée sur une entrée/sortie (*blocked*) mais n'a pas fini son exécution.
- Le seul cas de blocage considéré ici correspond à l'usage de verrous
- Les tâches *ready* sont répertoriés dans une « *ready queue* »: une ou plusieurs files d'attente.

# Les états d'une tâches

- Une tâche a quatre états:



Remarques:

- Les éléments en rouge sont ceux qu'on va modéliser explicitement avec l'annexe comportementale de AADL
- Ici *complete* fait référence à un mot clé de l'annexe comportementale AADL (voir slide suivant)

# Les états dans l'annexe comportementale

- **Initial**, correspond à l'état dans lequel se trouve le composant après initialisation
- **Final**, correspond à l'état dans lequel se trouve le composant après finalisation (retour d'un appel de fonction ou finalisation d'une tâche)
- **Complete**, utilisable pour les threads seulement: correspond à un état dans lequel le thread n'utilise pas la ressource d'exécution (préempté, attente passive, etc...)
- Une clause comportementale doit contenir un état initial et au moins un état final
  - ⌘ L'état final et initial peuvent être le même
- Un état *complete* correspond à l'état *suspended* vu précédemment
- Un état *non-complete* correspond à l'état *running*, *blocked* ou *ready* vu précédemment

# Les transitions dans l'annexe comportementale

- Un Etat source
- Une ensemble de conditions
  - ⌘ conditions d'exécution, correspondant à un switch/case dans l'exécution d'une fonctionnalité
    - Une condition peut porter sur le contenu d'un port, d'un paramètre, d'une donnée, d'une propriété, etc...
  - ⌘ conditions de dispatch, correspondant aux conditions de réveil d'un thread à partir d'un état *complete* donné
    - à mettre en regard de la propriété dispatch\_protocol du thread (Periodic, Sporadic, Aperiodic, Timed ou Hybrid)
- Un état cible
- Un ensemble d'actions
  - ⌘ Séquence d'actions pouvant contenir une structure de contrôle (if/then/else; while; do ... until) et des interactions avec l'environnement d'exécution
  - ⌘ Les conditions dans les structures de contrôle des actions sont similaires à des conditions d'exécution

# Interaction des clauses comportementales avec l'environnement d'exécution AADL

- Appel de sous-programmes: `sub!(v1, v2, r1);`
- Computation (10 ms); représente un calcul de 10 ms
- `p!` sur un port de sortie `p` permet d'envoyer le message contenu dans `p`; `p!(v)` permet d'envoyer le message `v` à travers `p`
- `p?` sur un port d'entrée permet de lire les messages contenus dans le buffer associé à `p`;
  - ⌘ Le contenu de la valeur renvoyée dépend du protocole associé au port (propriété *Dequeue\_Protocol*)
- `p'count` et `p'fresh` pour connaître l'état d'un buffer correspondant à un port `p`
  - ⌘ `p'count` renvoie le nombre d'éléments dans le buffer
  - ⌘ `p'fresh` renvoie `true` si la valeur a été mise à jour depuis sa dernière utilisation; `false` sinon



# Exemple d'utilisation de l'annexe comportementale

```
Thread controller_task
features
  P_i: in event port;
  P_o: out event port;
  A_i: in event port;
  A_o: out event port;
end controller_task;
```

```
thread implementation controller_task.impl
properties
  Dispatch_Protocol => Timed;
  Period => 1000 ms;
annex Behavior_Specification {**
  states
    idle: initial complete final state;
    detect_P: state;

  transitions
    idle -[on dispatch A]-> detect_P {
      while (P_i'count != 0) {P_i?};
      A_i?;
      A_o!;
      Computation(20 ms)};
    detect_P -[P_i'count > 0]-> idle {
      while (P_i'count != 0) {P_i?};
      };
    detect_P -[P_i'count = 0]-> idle {
      P_o!
      };
    idle -[on dispatch Period]-> idle;
  **};
end controller_task.impl;
```



# AADL pour la description de systèmes ARINC653

# Exemple d'utilisation: modéliser une architecture avionique

- Que doit-on représenter?

- Les partitions, et leur configuration (mémoire, partition windows, etc...)

```

SYSTEM IMPLEMENTATION monoProc.impl
SUBCOMPONENTS
    root_main_memory_inst : MEMORY root_main_memory.impl;
    train_inst : PROCESS train.impl;
    railway_inst : PROCESS railway.impl;
    root_inst : PROCESSOR root.impl;
END monoProc.impl;

PROCESSOR root
END root;

```

Représentation logique d'une partition (espace d'adressage)

```

PROCESSOR IMPLEMENTATION root.impl
SUBCOMPONENTS
    VP_train_inst : VIRTUAL PROCESSOR VP_train.impl;
    VP_railway_inst : VIRTUAL PROCESSOR VP_railway.impl;
PROPERTIES
    Scheduling_Protocol => (ARINC653);
    ARINC653::Module_Schedule => (
        [ Partition => reference(VP_train_inst );
          Duration => 250 ms;],
        [ Partition => reference(VP_railway_inst );
          Duration => 250 ms;] );
    ARINC653::Module_Major_Frame => 500 ms;
END root.impl;

```

Représentation physique d'une partition (machine virtuelle)

# Exemple d'utilisation: modéliser une architecture avionique

- Que doit-on représenter?

- ⌘ Les partitions, et leur configuration (mémoire, partition windows, etc...)
- ⌘ Les tâches de chaque partition, et leur configuration (périodique? Période, priorité, etc...)

**PROCESS IMPLEMENTATION** railway.impl  
**SUBCOMPONENTS**

```
trainPositionsCorrelation : THREAD trainPositionsCorrelation_Thread.impl;
viuTrainPositionEstimation : THREAD viuTrainPositionEstimation_Thread.impl;
trainSimulation : THREAD trainSimulation_Thread.impl;
delayed_trainDataIn : THREAD delayed_trainDataIn_Thread.impl;
```

**PROPERTIES**

```
Data_Size => 200KByte;
Code_Size => 200KByte;
Dispatch_Protocol => Periodic APPLIES TO trainPositionsCorrelation;
Priority => 3 APPLIES TO trainPositionsCorrelation;
Period => 600ms APPLIES TO trainPositionsCorrelation;
Stack_Size => 40KByte APPLIES TO trainPositionsCorrelation;
...
END railway.impl;
```

# Utilisation du modèle obtenu

- Spécification, documentation, conception
- Analyse à ce niveau de modélisation
  - ⌘ Ordonnançabilité gros grain (modèle de tâches, propriétés)
  - ⌘ Cohérence du dimensionnement (capacité/demande mémoire)
  - ⌘ Model-checking?
  - ⌘ Génération de code?

# Exemple d'utilisation: modéliser une architecture avionique

- Que doit-on représenter?

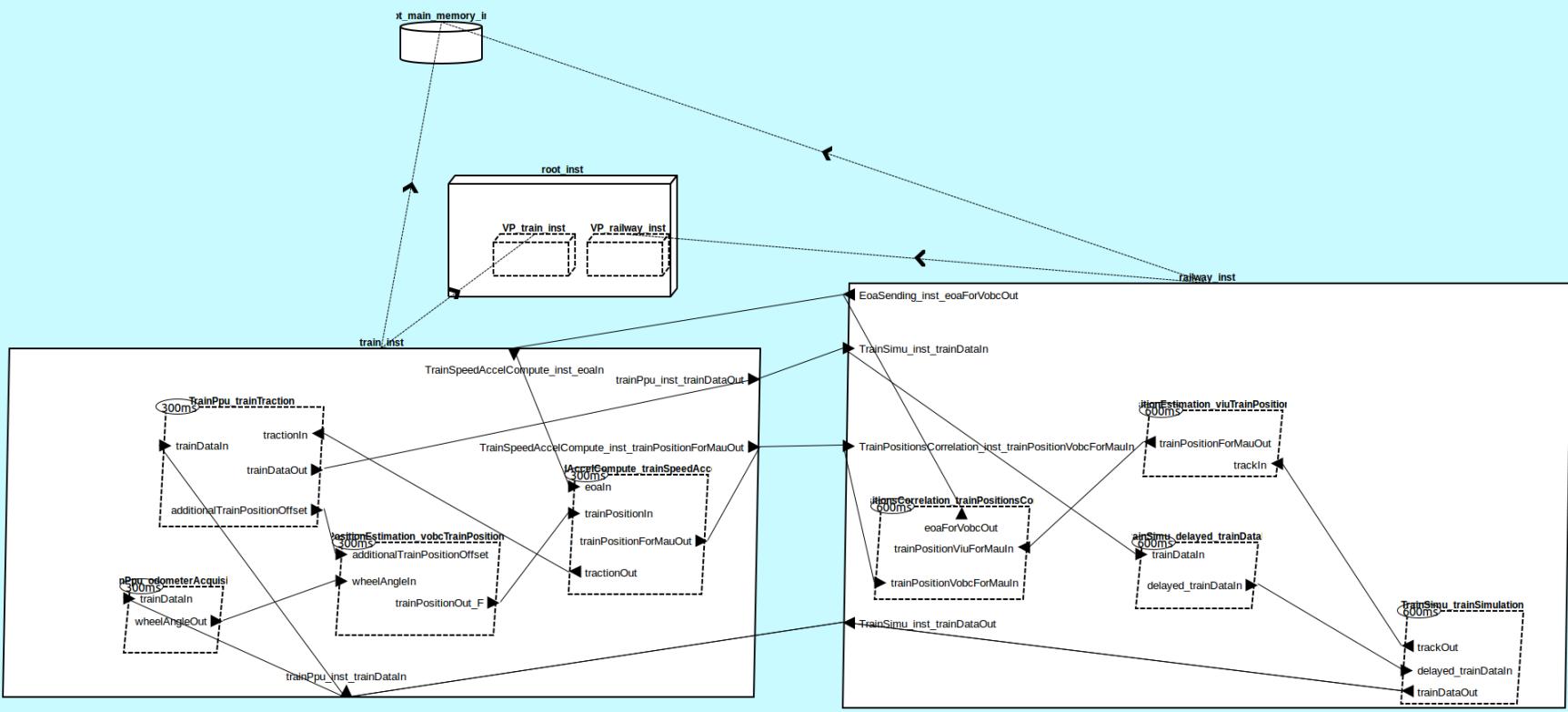
- ⌘ Les partitions, et leur configuration (mémoire, partition windows, etc...)
- ⌘ Les tâches de chaque partition, et leur configuration (périodique? Période, priorité, etc...)

- Que manque-t-il pour aller plus loin?

- ⌘ Les canaux de communication,
- ⌘ Le déploiement (mémoire et processeur)
- ⌘ Comportement des tâches...

Présenté précédemment

# Vue graphique d'un cas d'étude avionique (synthétique mais imprécise)



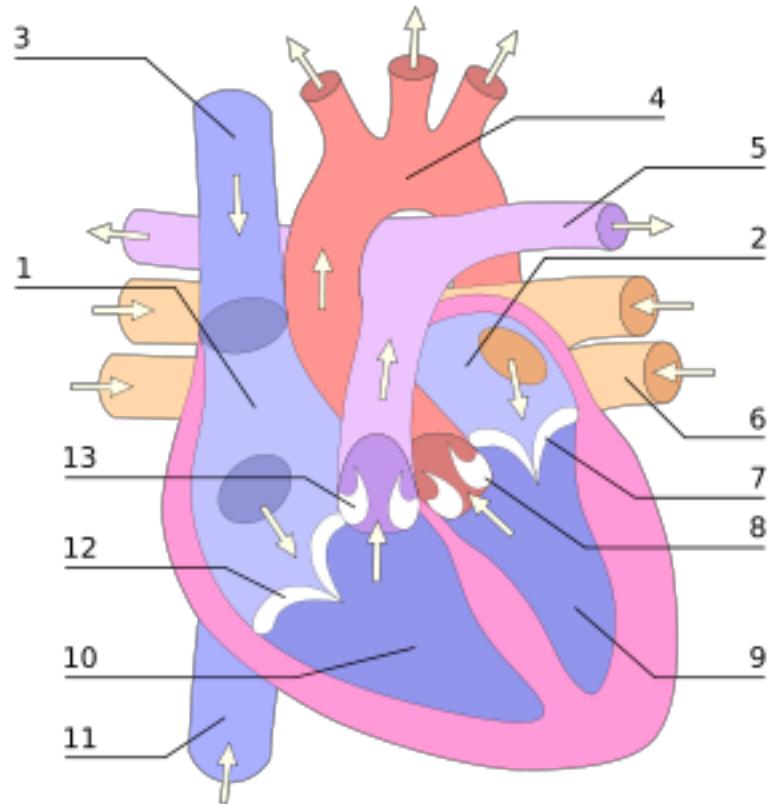
# Utilisation du modèle obtenu

- Spécification, documentation, conception
- Analyse supplémentaire à ce niveau de modélisation
  - ⌘ Cohérence du modèle (connexions des ports par ex.)
  - ⌘ Complétude de la configuration ARINC
  - ⌘ Génération de la configuration XML ARINC
- Que manque t-il pour aller plus loin?
  - ⌘ La modélisation du comportement des tâches



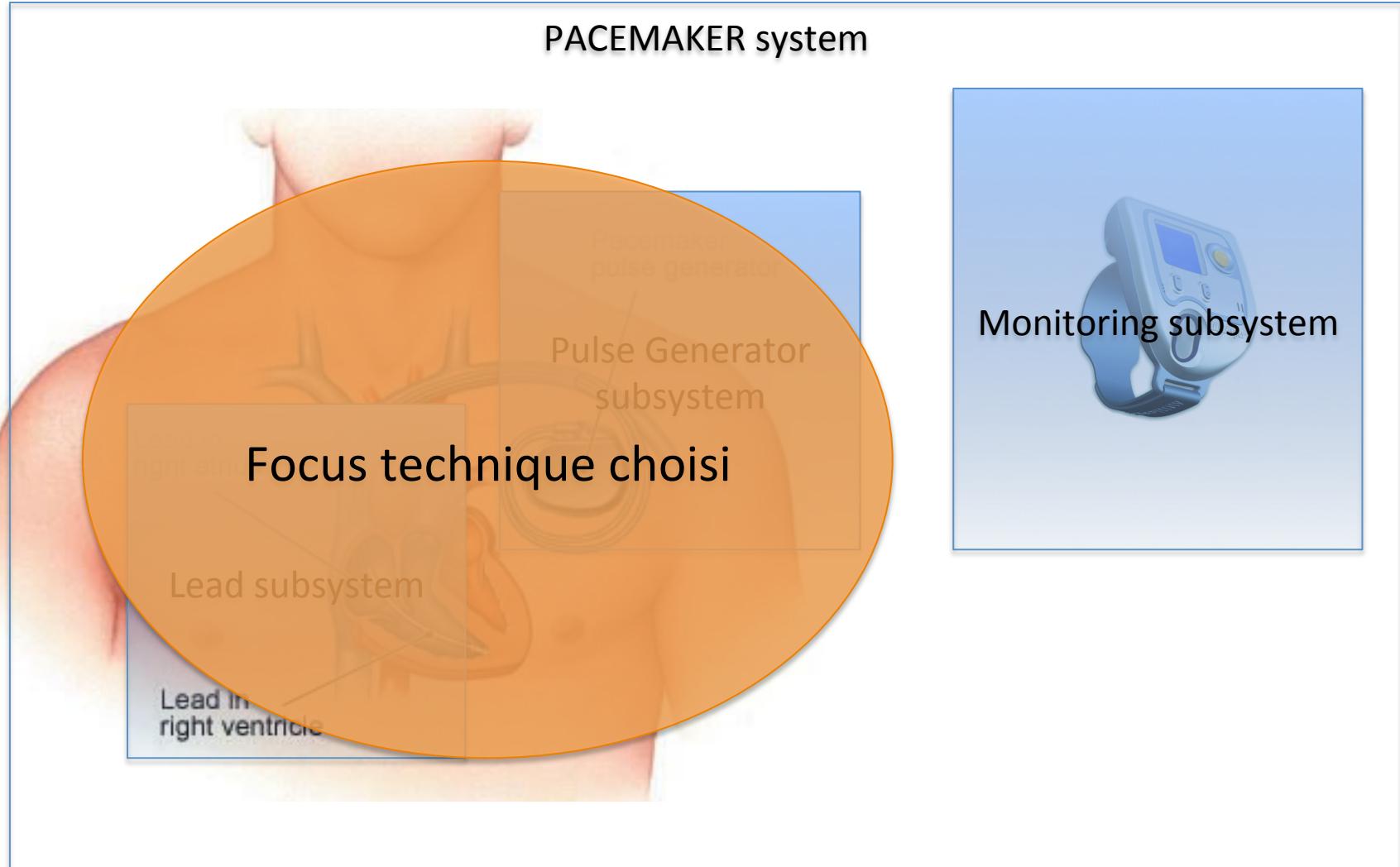
## Exemple complet: conception et vérification d'un pacemaker

# Bref rappel de biologie



1. Atrium droit
2. Atrium gauche
3. Veine cave supérieure
4. Aorte
5. Artère pulmonaire
6. Veine pulmonaire
7. Valve mitrale (atrio-ventriculaire)
8. Valve aortique
9. Ventricule gauche
10. Ventricule droit
11. Veine cave inférieure
12. Valve tricuspidale (atrio-ventriculaire)
13. Valve sigmoïde (pulmonaire)

# Décomposition systèmes/sous-systèmes





# Modes

The following bradycardia operating modes shall be programmable: Off, DDDR, VDDR, DDIR, DOOR, VOOR, AOOR, VVIR, AAIR, DDD, VDD, DDI, DOO, VOO, AOO, VVI, AAI, VVT and AAT.

	I	II	III	IV (optional)
Category	Chambers Paced	Chambers Sensed	Response To Sensing	Rate Modulation
Letters	O–None A–Atrium V–Ventricle D–Dual	O–None A–Atrium V–Ventricle D–Dual	O–None T–Triggered I–Inhibited D–Tracked	R–Rate Modulation

Table 2: Bradycardia Operating Modes

# Exigences temporelles

Parameter	A T	V T	A O	A I	V O	V I	V D	D O	D I	D O	A R	A O	V I	V V	V D	D O	D R	D I	D R
Lower Rate Limit	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Upper Rate Limit	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Maximum Sensor Rate										X	X	X	X	X	X	X	X	X	X
Fixed AV Delay						X	X	X	X						X	X	X	X	X
Dynamic AV Delay						X			X					X					X
Sensed AV Delay Offset								X											X
Atrial Amplitude	X		X	X			X	X	X	X	X					X	X	X	X
Ventricular Amplitude		X			X	X	X	X	X	X			X	X	X	X	X	X	X
Atrial Pulse Width	X		X	X			X	X	X	X	X					X	X	X	X
Ventricular Pulse Width		X			X	X	X	X	X	X			X	X	X	X	X	X	X
Atrial Sensitivity	X			X				X	X		X						X	X	
Ventricular Sensitivity		X			X	X		X	X				X	X			X	X	
VRP		X			X	X		X	X				X	X			X	X	
ARP	X			X				X	X		X						X	X	
PVARP	X			X				X	X		X						X	X	
PVARP Extension						X			X						X			X	
Hysteresis			X		X				X	X		X						X	
Rate Smoothing		X		X	X			X		X		X		X	X			X	
ATR Duration						X			X						X			X	
ATR Fallback Mode						X			X						X			X	
ATR Fallback Time						X			X						X			X	
Activity Threshold										X	X	X	X	X	X	X	X	X	X
Reaction Time										X	X	X	X	X	X	X	X	X	X
Response Factor										X	X	X	X	X	X	X	X	X	X
Recovery Time										X	X	X	X	X	X	X	X	X	X

Table 6: Programmable Parameters for Bradycardia Therapy Modes

*LRL: rythme minimum de battement*

*URL: rythme maximum de battement*

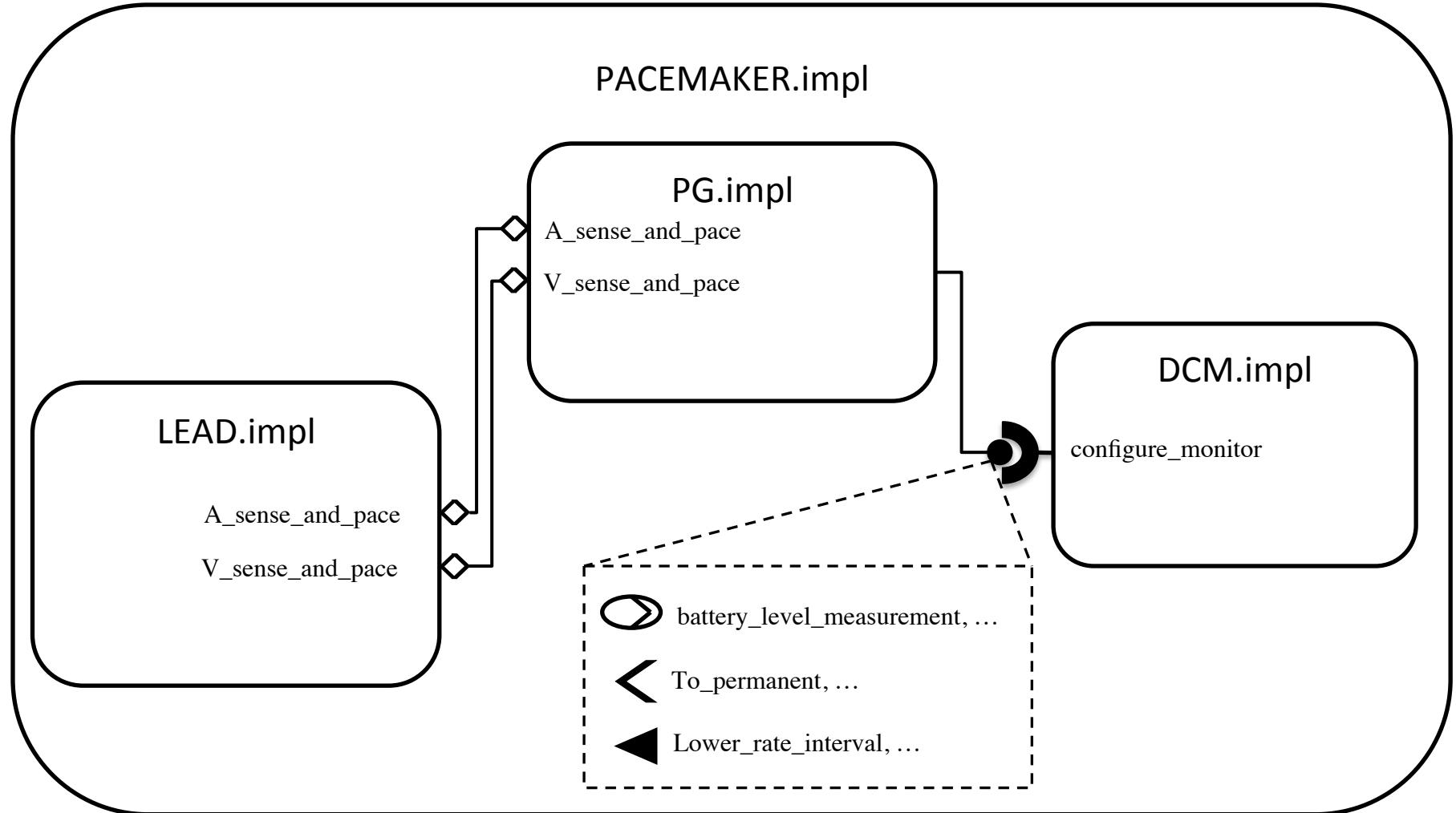
*AV: durée séparant un battement dans l'oreillette d'un battement dans le ventricule*

*ARP: durée minimale entre deux battements dans l'oreillette*

# Démarche de modélisation

- Décomposition système/sous-système
- Identification des composants logiciels et de leurs interfaces
- Identification des composants matériels et de leurs interfaces
- Description du comportement des composants
  - ⌘ Synchronisation du générateur de pulsation avec le fonctionnement naturel du cœur
  - ⌘ Modes de fonctionnement et transitions de mode

# Modèle AADL: système global

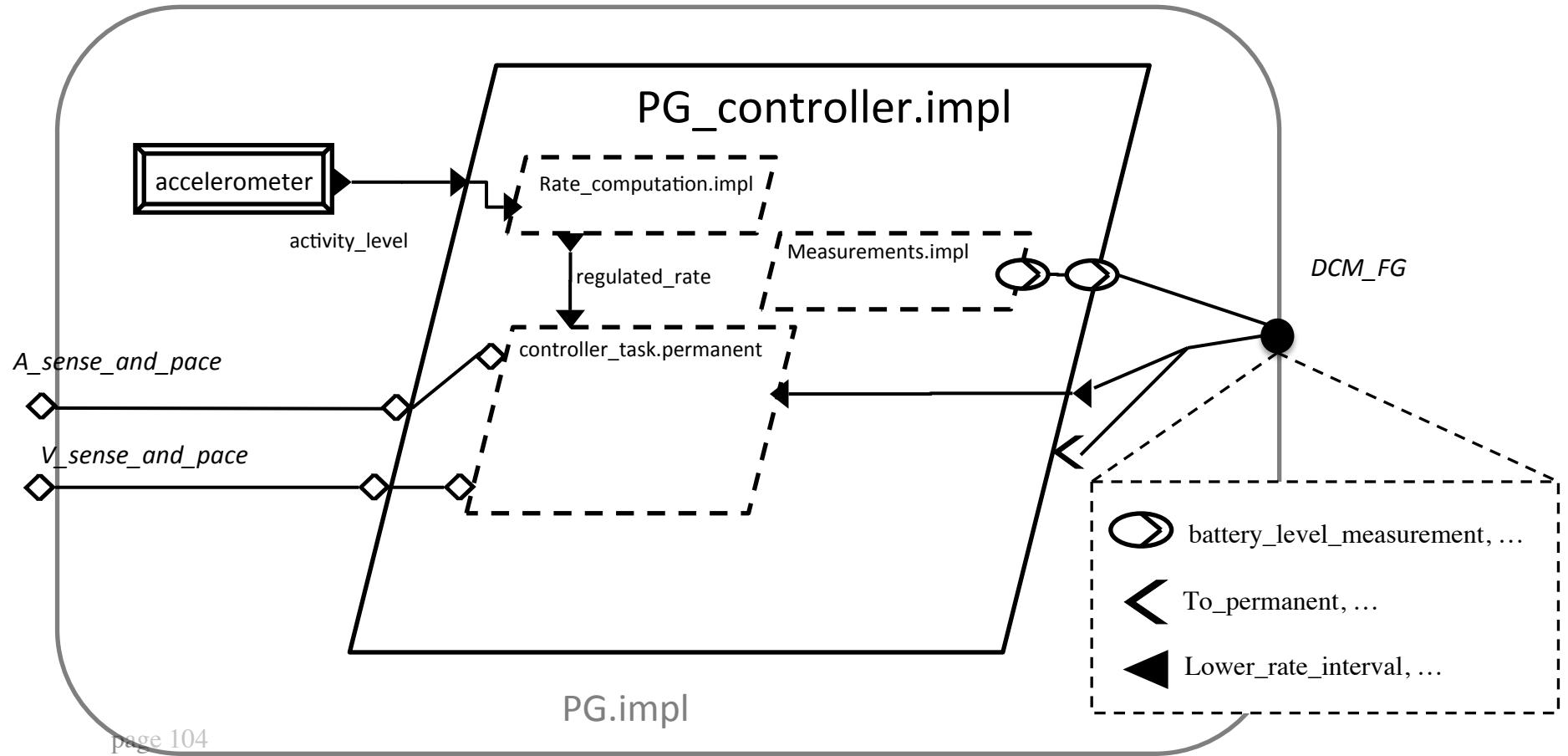


# Identification des composants logiciels et de leurs interfaces

- Pulse Generator
  - ⌘ Un processus de contrôle
    - Entrées: détection de battements (ventriculaire/atriale)
    - Sorties: stimulation (ventriculaire/atriale)
  - ⌘ Une de tâches de contrôle
    - Mêmes entrées/sorties
  - ⌘ Une ou plusieurs tâches de configuration/supervision pour s'interfacer avec le DCM
- Ensembles de connections triviales:
  - ⌘ Périphériques de capture → entrées du processus de contrôle
  - ⌘ Sorties du processus de contrôle → périphériques de stimulation

# Composants logiciels

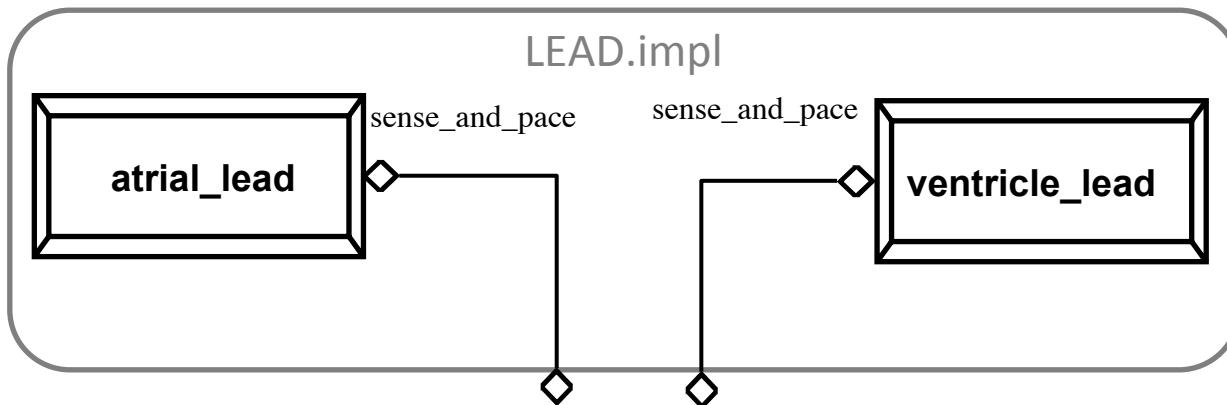
## PG: processus de contrôle



# Composants du sous-système LEAD

## ■ Leads

- Capteurs (ventriculaire/atriale) des battements naturels
- Stimulation (ventriculaire/atriale) du cœur



# Modèle AADL: Lead system

```

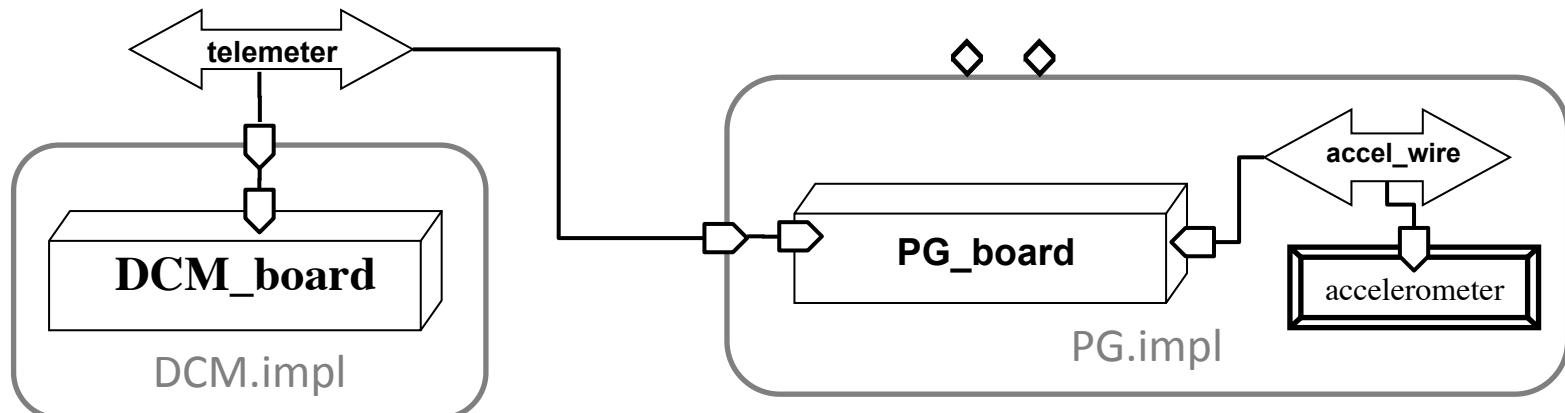
device compartment_lead
  features
    sense_and_pace: in out event port;
end compartment_lead;

system implementation LEAD.impl
  subcomponents
    atrial_lead: device compartment_lead;
    ventricle_lead: device compartment_lead;
  connections
    atrial_lead. sense_and_pace -> A_sense_and_pace;
    ventricle_lead. sense_and_pace -> V_sense_and_pace;
end LEAD.impl;

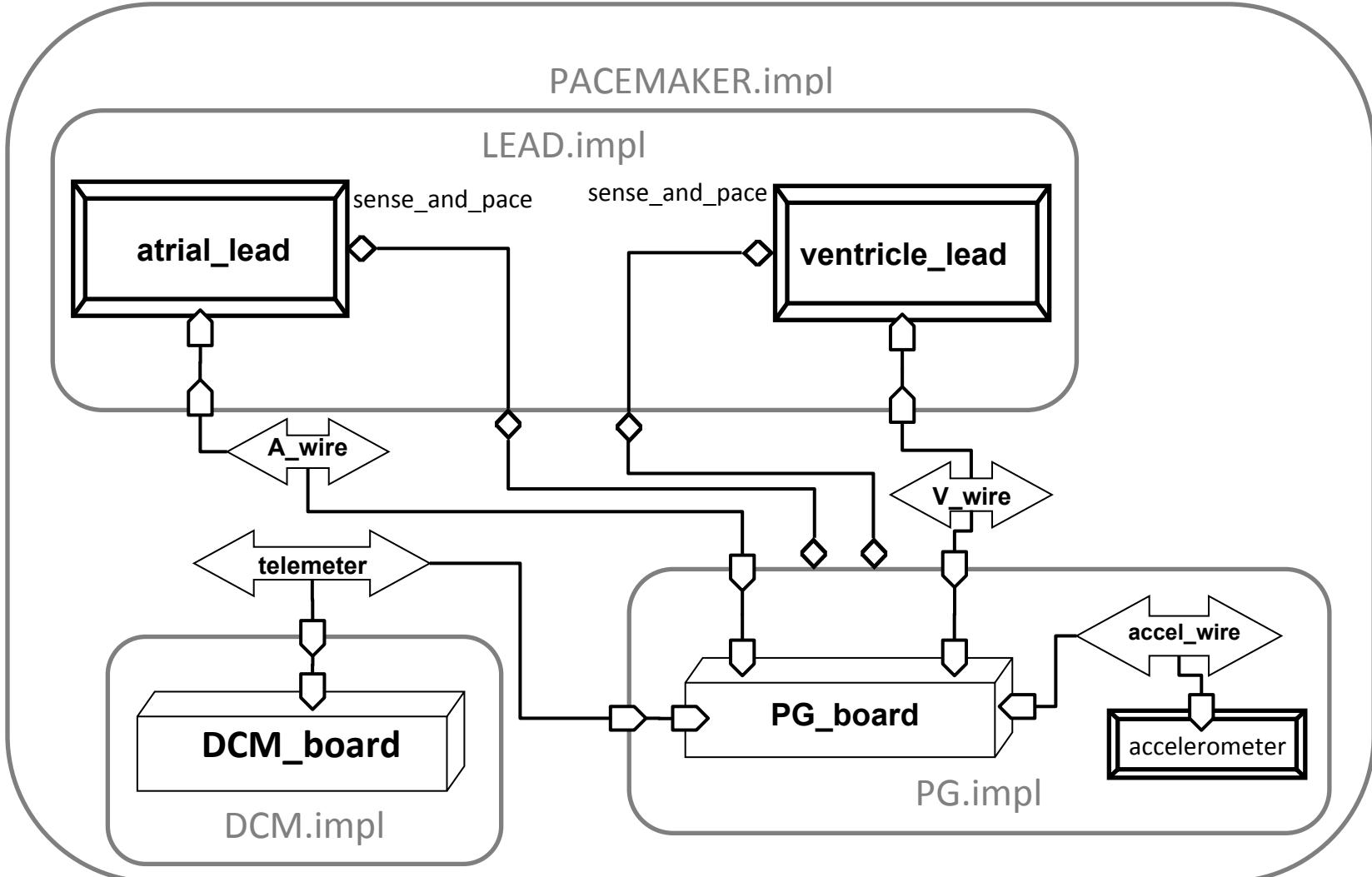
```

# Plate-forme d'exécution DCM et PG

- Pulse Generator
  - ⌘ Resource d'exécution des fonctionnalités de régulation cardiaque
  - ⌘ Acceleromètre
- DCM
  - ⌘ Resource d'exécution des fonctionnalités de régulation cardiaque
- Medium de communication entre DCM et PG



# Composants matériel



# Description du comportement

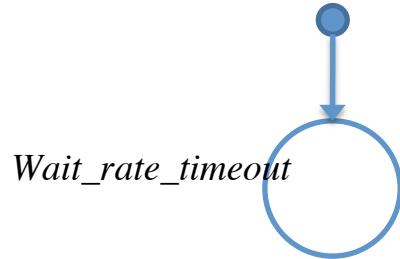
- Prédominance des modes de fonctionnement dans la définition du comportement
- Dont la portée peut-être décomposé en
  - ⌘ Impact des changements de mode sur la structure de l'architecture (composants/interfaces/connections...)
  - ⌘ Impact des changements de mode sur le comportement (périodes/dispatch\_protocole/fonctionnalités...)
- Enfin, à mode constant, certains éléments de l'architecture sont “programmable” ou “variable”
  - ⌘ Exp. le rythme de battement (période d'une fonctionnalité) dépend des mouvements de l'utilisateur (données fournis par un accéléromètre)

# Ordonnancement des tâches

- *Scheduling\_Protocol, Dispatch\_Protocol, et éventuellement Period et Priority*
  - ⌘ PG\_Controller
  - ⌘ Rate\_Computation
  - ⌘ Measurements
- Attention à maintenir la cohérence entre les propriétés
  - ⌘ Ex: pas de période pour une tâche apériodique
  - ⌘ Ex: pas de tâche apériodique si la politique d'ordonnancement est RMS
  - ⌘ Ex: Period ne peut prendre qu'une valeur entière constante

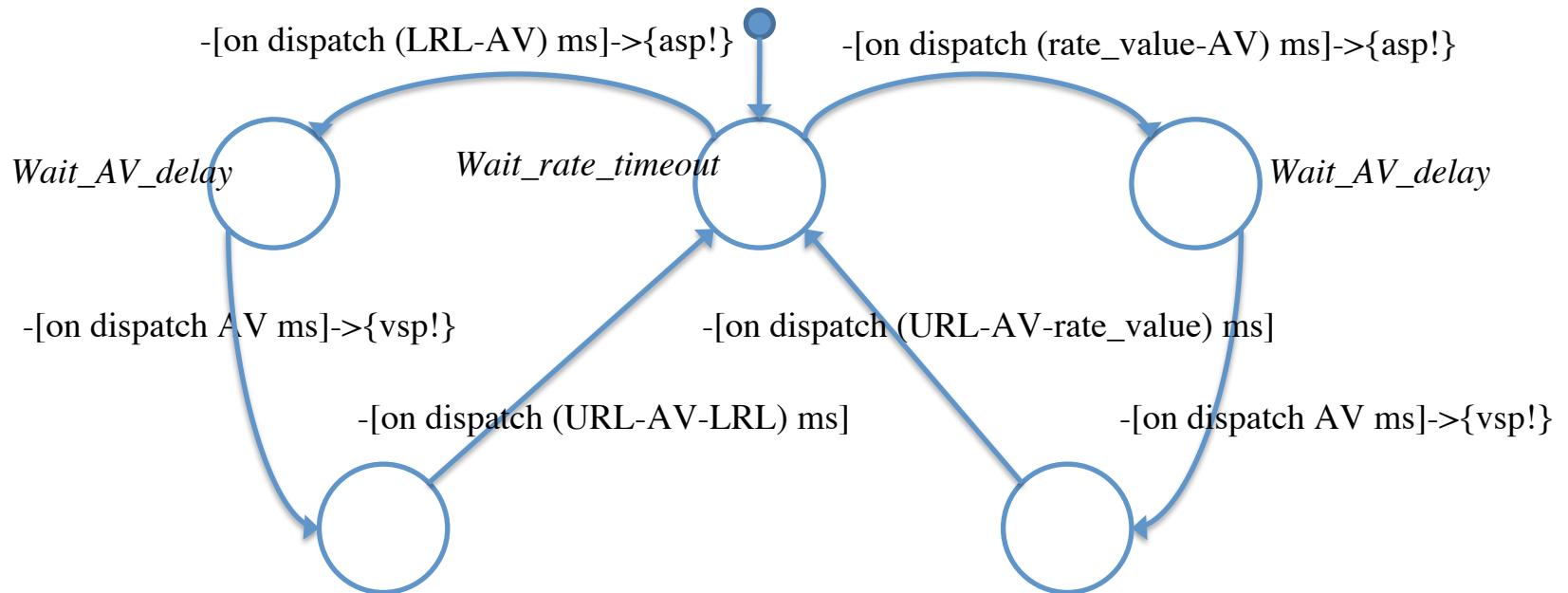
# Comportement de la tâche de contrôle dans le mode DOOR

- **Contraintes temporelles:**
  - ⌘ LRL, AV, URL, et rate\_value



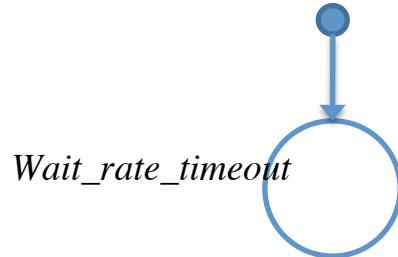
# Comportement de la tâche de contrôle dans le mode DOOR

- **Contraintes temporelles:**
  - ⌘ LRL, AV, URL, et rate\_value



# Comportement de la tâche de contrôle dans le mode AAI

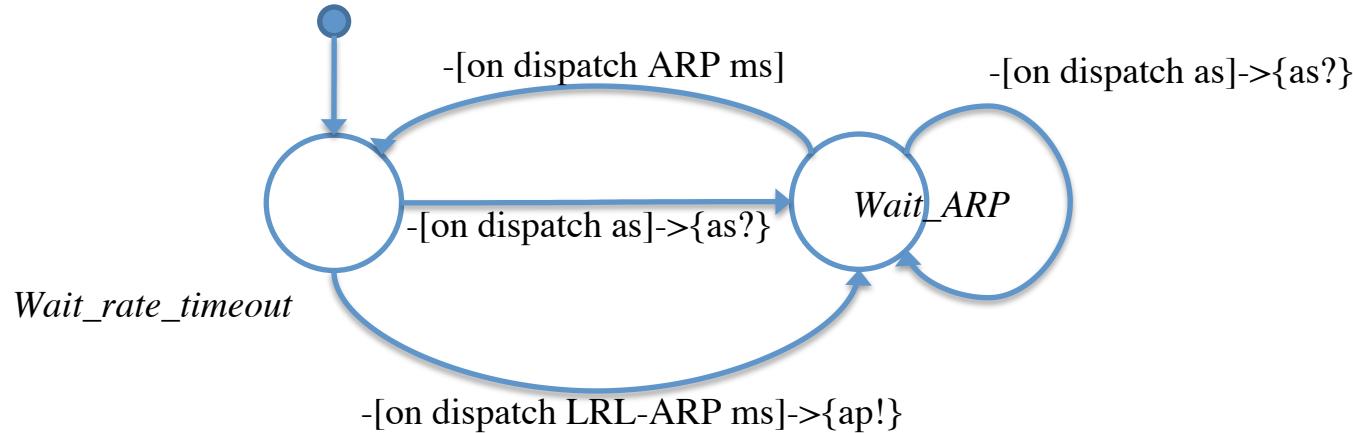
- **Contraintes temporelles:**
  - ⌘ LRL, ARP, et URL



# Comportement de la tâche de contrôle dans le mode AAI

- **Contraintes temporelles:**

- ⌘ LRL, ARP, et URL



# Défauts de la conception: partie comportementale

- La solution proposée permet-elle d'assurer le respect des exigences temporelles?
- Quelques soient les conditions d'utilisation?

# Défauts de la conception: partie comportementale

- La solution proposée permet-elle d'assurer le respect des exigences temporelles?
- Quelques soient les conditions d'utilisation?
  - ⌘ En cas de modification des valeurs LRL, etc... sur les data ports d'un thread ???
- Quelle solution à ce problème?

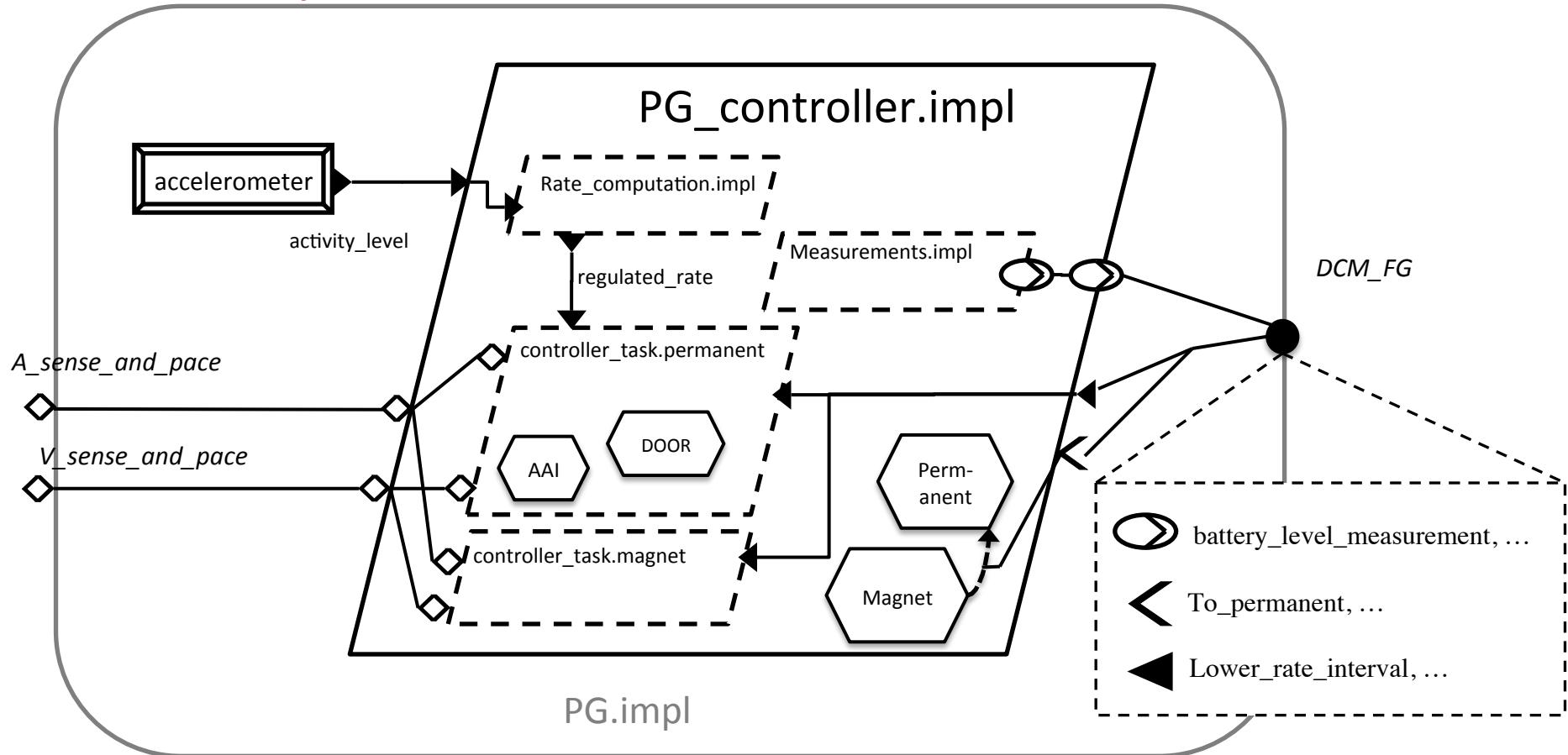
# Modes de fonctionnement

## Présence importante des modes de fonctionnement

- ⌘ 5 modes principaux: Permanent/Temporary/Pace-Now/  
Magnet/Power-On Reset
- ⌘ + de 30 Sous-modes:
  - Permanent: Off, DDDR, VDDR, DDIR, DOOR, VOOR, AOOR, VVIR,  
AAIR, DDD, VDD, DDI, DOO, VOO, AOO, VVI, AAI, VVT and AAT.
  - Temporary: OVO, OAO, ODO, and OOO.
  - Pace-Now: VVI.
  - Magnet: AXXX, VXXX, DXXX en fonction du mode source; et sous-  
modes BOL, ERT, ERN.
  - Power-On reset: VVI.

# Modélisation des modes de fonctionnement

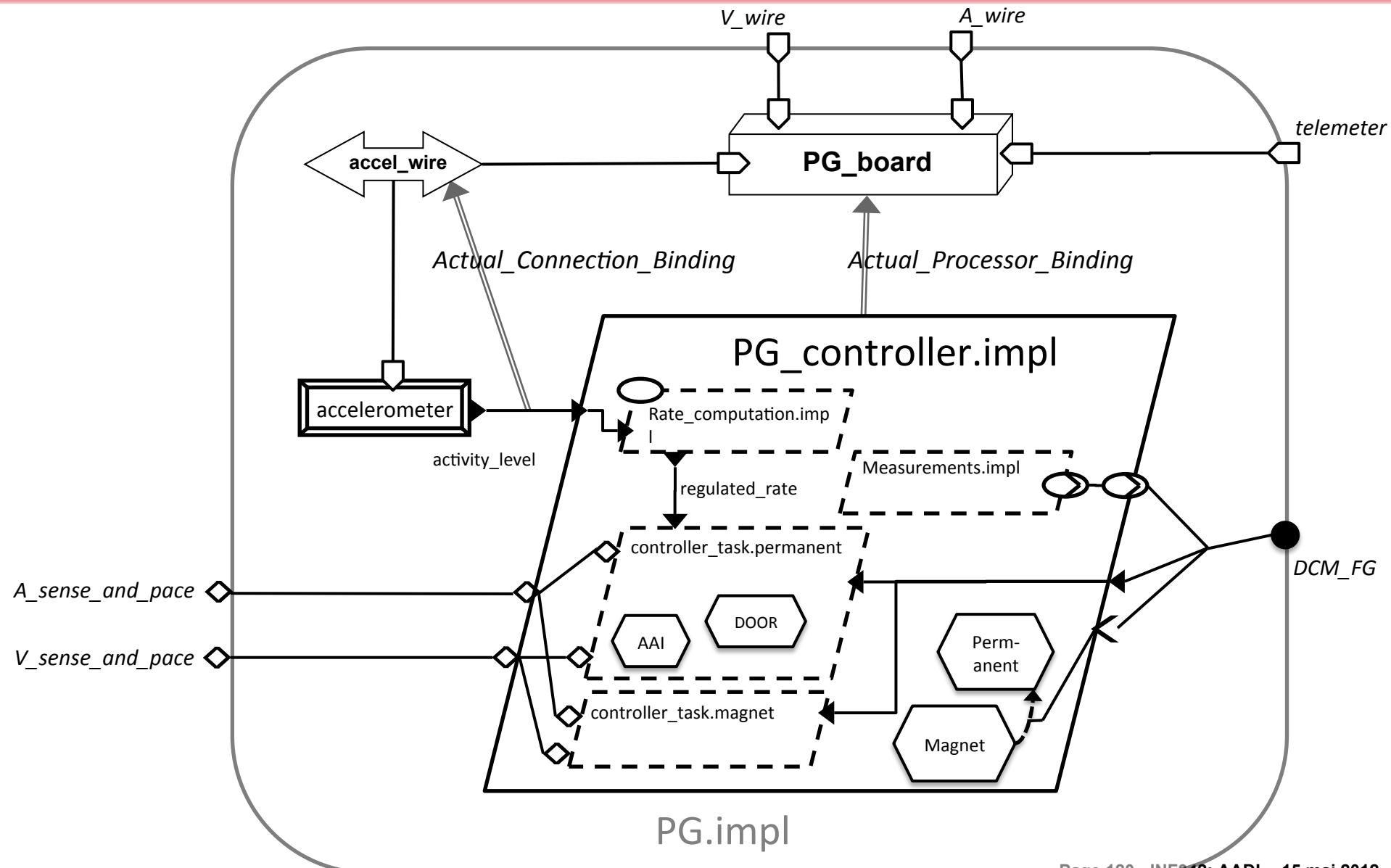
- PG: processus de contrôle



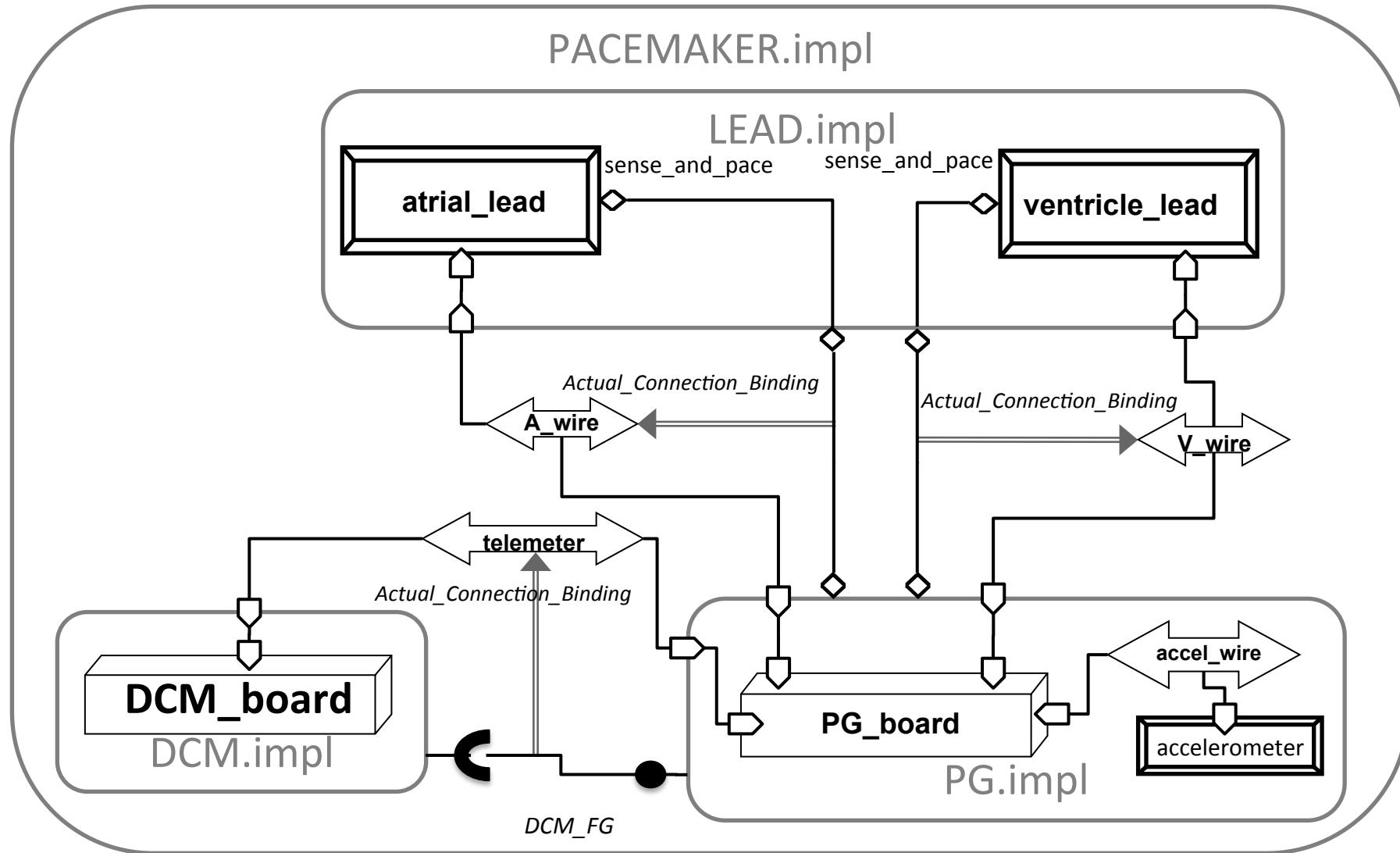
# Défauts de la solution

- La solution proposée permet-elle d'assurer le respect des exigences temporelles?
- Quelques soient les conditions d'utilisation?
  - ⌘ En cas de changement de mode ?
- Quelle solution à ce problème?

# Architecture du sous-système PG



# Architecture matérielle complète



# Conclusion concernant ce cas d'étude

- Le PACEMAKER est un système beaucoup plus complexe qu'il paraît être de prime abord
- Les informations contenues dans la spécification système sont très éloignées de la conception finale
- La modélisation permet de lever des ambiguïtés de fonctionnement à condition d'avoir
  - ⌘ une expertise forte dans le langage de modélisation choisi
  - ⌘ une expertise suffisante dans le système à concevoir (donc son domaine d'application)
- Quelle exploitation pour ces modèles?
  - ⌘ Communication
  - ⌘ Analyse et vérification
  - ⌘ Génération de code

# Conclusions

- **Modélisation concrète**
  - ⌘ dernière phase avant la génération/déploiement du système
  - ⌘ précision dans la modélisation
  - ⌘ exploitation pour la vérification et la génération de prototypes
- **AADL offre une grande souplesse de modélisation**
  - ⌘ degré de modélisation selon les besoins
  - ⌘ peut être utilisé comme langage fédérateur
    - exploitation par plusieurs outils différents
- **Pour aller plus loin**
  - ⌘ Syntaxe AADL complète
  - ⌘ Exemples de modèles AADL:
  - ⌘ Site web AADL:
- **Outils**
  - ⌘ <http://penelope.enst.fr/aadl>
  - ⌘ <http://beru.univ-brest.fr/~singhoff/cheddar/index-fr.html>