AADL: Architecture Analysis & Design Language

Etienne Borde, Gilles Lasnier, Bechir Zalila,
Laurent Pautet, Thomas Vergnaud
{nom.prenom}@telecom-paristech.fr

Introduction et généralités

Contexte

- 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

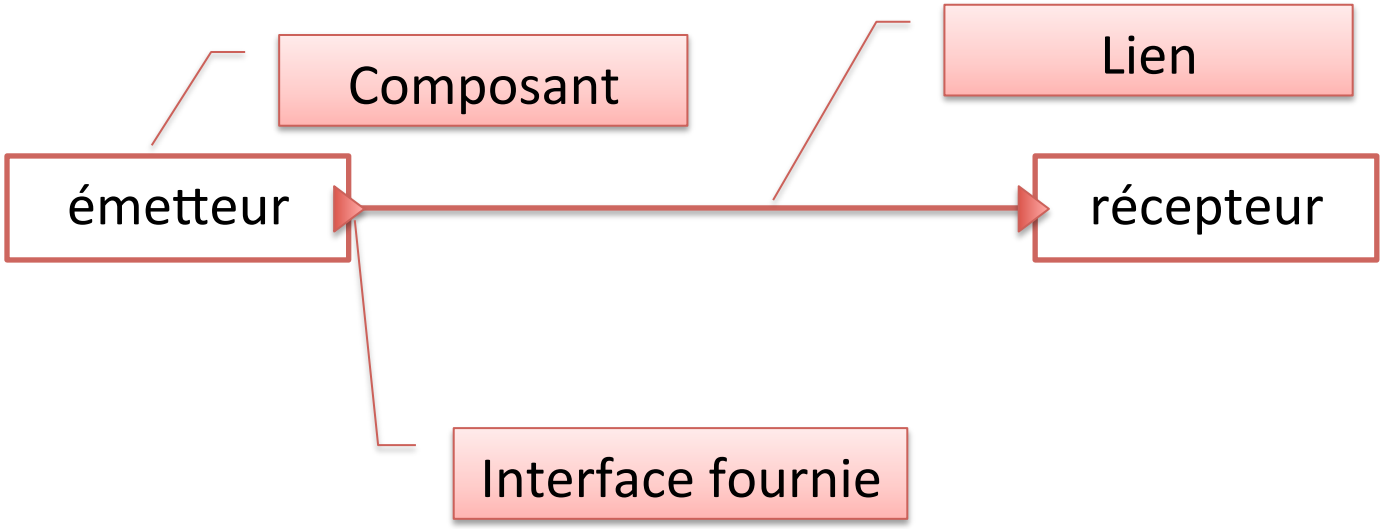
Modélisation d'architectures

- Modèle: représentation abstraite d'une architecture logicielle ou matériel dans le but de faciliter la compréhension et l'analyse d'un système.

- Modèles fonctionnels/architecturaux
  - ⌘ On aborde que les modèles architecturaux dans ce cours

Les langages de description d'architecture (ADL) - 1

- 3 éléments principaux :
  - les composants (ensemble d'interfaces requises et fournies)
  - les liens entre les composants (chemins d'échanges d'information)
  - une sémantique associée à ces éléments (comportement correspondant à une spécification; un langage)



Sémantique : émetteur transmet une donnée a récepteur

Les langages de description d'architecture (ADL) - 2

- ADL formels
  - Formalisent la description du fonctionnement d'un système
  - S'intègrent mal dans une démarche de génération automatique
  - Wright, Rapide
- ADL concrets
  - Décrivent l'architecture attendue afin de la générer automatiquement
  - UML 2, ***AADL***
- ADL restreints
  - Décrivent l'assemblage de composants logiciels sans sémantique opérationnelle forte
  - ArchJava, Fractal

Les langages de description d'architecture (ADL) - 3

- Critères de choix :
  - ⌘ Générique / Spécifique à un domaine
  - ⌘ Standardisé / Ouvert / Propriétaire
  - ⌘ Graphique / Textuel

	Avantages	Inconvénients
Générique	Extensible à plusieurs domaines	Sémantique faiblement définie
Standardisé	Interopérabilité	Difficile à adapter
Ouvert	Facile à adapter	Pas d'interopérabilité
Textuel	Pas de dépendance aux outils d'édition	Peu synthétique
Graphique	Facile à lire, à partager	Dépendance aux outils, limite le nombre d'information disponible

Les langages de description d'architecture (ADL) - 4

- Critères de choix :
  - ⌘ Générique / Spécifique à un domaine
  - ⌘ Standardisé / Ouvert / Propriétaire
  - ⌘ Graphique / Textuel

FractalADL	D&C (OMG)	UML (OMG)	AADL (SAE)
Générique	Générique	Générique	Spécifique extensible
Ouvert	Standard (OMG)	Standard (OMG)	Standard (SAE)
Textuel (XML)	Textuel (XML)	Graphique	Textuel; correspondance graphique

Usage des modèles dans l'industrie?

- Surtout UML, mais DSLs (Domain Specific languages) dans le domaine de l'embarqué
- Langage trop complexes ➔ apprentissage long et couteux
- Objectifs:
  - ⌘ Spécification,
  - ⌘ Documentation,
  - ⌘ Conception,
  - ⌘ Analyse,
  - ⌘ Génération de code

Les principes de base d'AADL

Pourquoi AADL?

- ADL spécifique aux systèmes temps-réels:
  - Certaines propriétés prédéfinies décrivent le comportement du système (politique d'ordonnancement, etc…)
  - Certaine propriétés permettent de représenter les allocation des ressources (mémoire, temps, etc…)

- AADL facilite ainsi la conception de tels systèmes:
  - Automatise la production de l'application (génération de code)
  - Facilite l'analyse (test d'ordonnançabilité, vivacité, etc...)

AADL: objectif principal et moyens

- Faciliter la conception (*implantation et analyse*) des systèmes temps-réels distribués
  - ⌘ Définit une sémantique aussi précise (et concrète) que possible pour l'ensemble des éléments du langage
  - ⌘ Ne pouvant couvrir l'ensemble des exigences de conception du domaine, AADL propose des mécanismes d'extension (propriétés, annexes, etc…)
  - ⌘ Propose trois niveau de modélisation:
    1. Système
    2. Logiciel
    3. Plate-forme d'exécution

Architecture Analysis & Design Language

- 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

AADL (http://www.aadl.info)

- Déjà utilisé par de grands projets
  - COTRE (Airbus)
  - ASSERT (ESA, EADS, ENST, INRIA, etc…)
  - Topcased (Airbus, CNES, ENST, etc…)
  - Flex-eWare (Thales, ENST, LIP6, CEA, INRIA, etc…)
  - …
- Quelques outils disponibles
  - OSATE : outil de référence pour Eclipse (SEI/CMU)
    - syntaxe textuelle et graphique
    - vérifications syntaxiques et sémantiques générales
    - plusieurs extensions pour la vérification d'architectures
  - STOOD : outil de modélisation basé sur la méthode HOOD (Ellidiss)
    - outil graphique (syntaxe UML & HOOD)
    - générateur de code pour des applications monolithiques
  - RAMSES: suite d'outils pour générer des applications (ENST)
    - transformations de modèles AADL → AADL
    - plusieurs générateurs d'applications (Ada et C)
  - Cheddar : outil d'analyse
    - Tests de faisabilité et simulation de l'ordonnancement
    - dimensionnement mémoire

Les composants AADL et leur composition

Caractéristiques générales

- Composants concrets
  - Catégorie (Thread, data, bus, processor, etc…)
  - Type : définition des interfaces (ports)
  - Implémentation : définition de la structure interne des composants (sous-composant, code, spec. comportementale, etc…)
  - Connections : relient les interfaces de sous-composants
- 1 Catégorie -> N types; 1 type -> N implémetations
- Des propriétés (prédéfinies ou définies par le concepteur) peuvent être associées à chaque élément de modélisation (composants, ports, connections,…)
- Langage descriptif: les éléments peuvent être spécifiés dans n'importe quel ordre

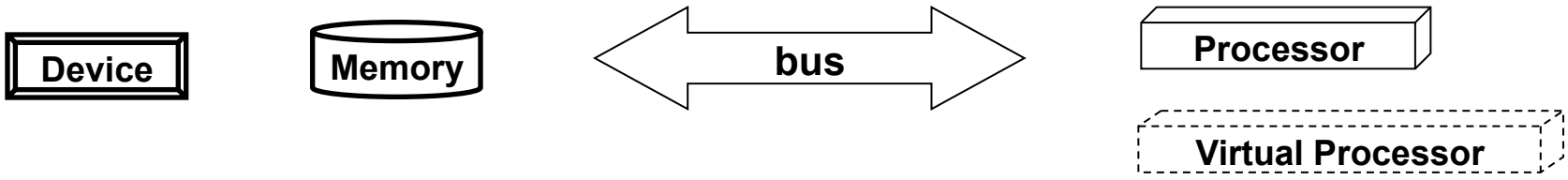
Catégories de composants

- Éléments de base d'une description architecturale

- Plusieurs ensembles de catégories
  - Plate-forme d'exécution (*execution platform components*)
  - logiciels (*software components*)
  - systèmes (*system composition*)

Description de la plate-forme d'exécution

- Plusieurs catégories
  - processor : micro-processeur + ordonnanceur
  - Virtual processor: unité d'exécution logique qui peut partager une resource physique
  - memory : disque dur, mémoire vive, etc.
  - bus : réseau, etc.
  - device : composant dont on ignore la structure interne
- Un processor modélise processeur + noyau contenant entre-autres un ordonnanceur.
- Un device sert typiquement à modéliser un capteur + le pilote de ce capteur

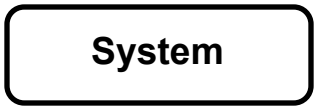


Description du logiciel

- Plusieurs catégories
  - ⌘ thread : fil d'exécution (ou thread dans les noyaux)
  - ⌘ data : structure de données
  - ⌘ process : processus, un espace mémoire pour l'exécution des threads qu'il contient
  - ⌘ thread group : crée une hiérarchie dans les threads
  - ⌘ 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)
- Peut contenir :
  - ⌘ system
  - ⌘ processor, memory, device, bus
  - ⌘ process, data
- MAIS PAS :
  - ⌘ thread
  - ⌘ thread group
  - ⌘ subprogram



Exemple de modèle AADL

```
package nxt_use_case
public

with Data_Model;
with OSEK, Generation_Properties;

system nxt
end nxt;

system implementation nxt.Impl
subcomponents
  nxt_cpu  : processor arm.impl;
  nxt_mem  : memory nxt_mem.impl;
  PROC     : process Proc.Impl;
…
end nxt.Impl;
```

```
processor arm
end arm;

processor implementation arm.impl
properties
  Scheduling_Protocol => (RMS) ;
end arm.impl;

process implementation Proc.Impl
Subcomponents
path: thread PathComputationThread.impl;
Find_Path: thread FindNextDirection.impl;
S: data Robot_state;
…
end Proc.Impl;

end nxt_use_case;
```

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: paramètres

- Pour des sous-programmes, les ports sont des paramètres (**`parameters`**)
- Un paramètre s'utilise comme un port de donnée
  - ⌘ data port
  - ⌘ event data port
- Les paramètres peuvent être `in`, `out` ou `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 

- 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 accès aux composants - 2

- On exprime ainsi l'obligation de brancher un composant avec un autre pour obtenir un système cohérent
  - Ex. un processor, une memory ou un device doivent être connectés aux autres composants matériels par l'intermédiaire d'un bus

- Cela permet également de représenter l'accès à des données partagées (pas forcément protégées)

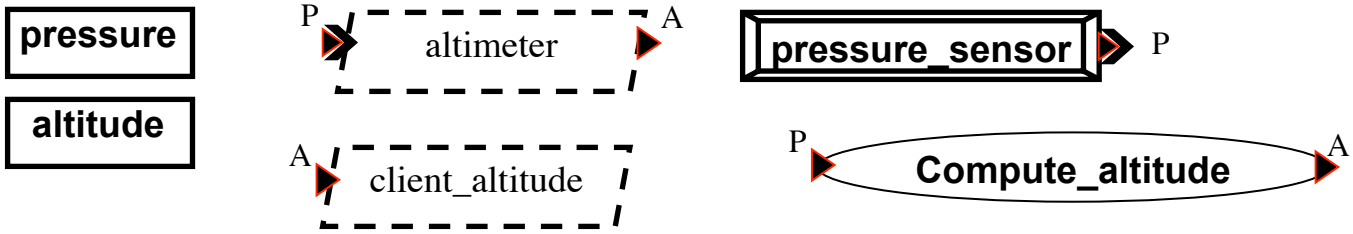
Features: les groupes de ports

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

Exemple de features

```
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

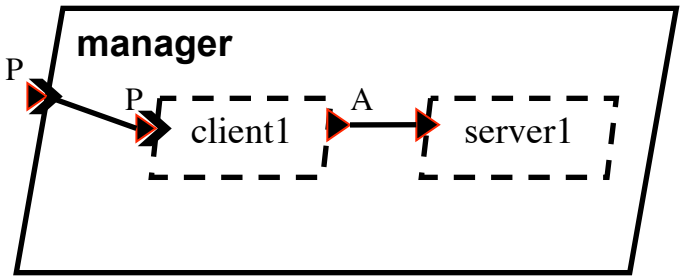
Les connexions des composants AADL

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 (1)

```
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 de connexion (2a)

```
data signal
end signal;

port group signal_DB9
features
  CD  : in  data port signal;   -- carrier detection
  RD  : in  data port signal;   -- data reception
  TD  : out data port signal;   -- data transmission
  DTR : out data port signal;   -- ready to transmit data
  DSR : in  data port signal;   -- ready to send data
  RTS : out data port signal;   -- transmision request
  CTS : in  data port signal;   -- ready for transmission
  RI  : in  data port signal;   -- reception indicator
end signal_DB9;

port group signal_DB9_inverse inverse of signal_DB9
end signal_DB9_inverse;
```

Exemple de connexion (2b)

```
system serial_card
features
  plug : port group signal_DB9;
end serial_card;

system serial_wire
features
  plug : port group signal_DB9_inverse;
end serial_wire;

system global
end global;

system implementation global.basic
subcomponents
  card : system serial_card;
  wire : system serial_wire;
connections
  port group card.plug -> wire.plug;
end global.basic;
```

Le rôle des propriétés dans un modèle AADL

Les propriétés en AADL (1)

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

- Une propriété permet d'associer une valeur d'un certain type (ou non typée) à un identifiant du modèle.
  - la norme prévoit un ensemble de ***propriétés standard***
  - il est possible de définir de nouvelles propriétés dans des ensembles de propriétés (property sets)

- Une propriété peut ne s'appliquer qu'à un ensemble de catégories d'éléments (p.ex. les processeurs)

Les propriétés en AADL (2)

- Le type d'une propriété peut être
  - 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`**
- **`applies to`** spécifie à quels types d'entité s'applique la propriété

Les propriétés en AADL (3)

- Les types de propriété peuvent s'appuyer sur des types existants:
  - Period: **`inherit Time`**
- Peuvent s'ajouter ou remplacer la valeur définie dans un composant père
  - extension de composant
- Peuvent être déclarées
  - dans un composant,
  - au niveau de la déclaration d'un sous-composant, une connexion, etc…
  - dans un composant père et s'appliquer à un de ses sous-élément (sous-composant, feature, appel à un sous-programme…)
- les propriétés permettent (entre autres) d'indiquer le déploiement des composants logiciels sur les composants matériels

Exemples de propriétés prédéfinies

```
Supported_Queue_Processing_Protocols :
  type enumeration (FIFO, <project_related>);

Queue_Processing_Protocol :
  Supported_Queue_Processing_Protocols => FIFO
    applies to (event port, event data port, subprogram);

Source_Text : inherit list of aadlstring
  applies to (data, port, subprogram, thread, thread group,
              process, system, memory, bus, device, processor,
              parameter, port group);

Max_Thread_Limit : constant aadlinteger => <project_related>;

Thread_Limit : aadlinteger 0 .. value (Max_Thread_Limit)
  => value (Max_Thread_Limit) applies to (processor);
```

Exemples d'associations de propriétés

```
processor a_processor
end a_processor;

processor implementation a_processor.simple
properties
  Thread_Swap_Execution_Time => 0ms .. 10 ms;
end a_processor.simple;

process a_process
end a_process;

system global
end global;

system implementation global.simple
subcomponents
  processor1 : processor a_processor.simple {Thread_Limit => 3;};
  process1   : process a_process;
properties
  Actual_Processor_Binding => reference processor1 applies to process1;
end a_process.simple;
```

Exemple d'ensemble de propriétés

```
property set our_properties is
  pressure     : type aadlinteger units (Pa, HPa => 100 * Pa);
  --  Définition des unités Pascal et Hecto-Pascal

  pressure_max : pressure applies to (device);
end our_properties;

system a_system
end a_system;

device a_sensor
end a_sensor;

system implementation a_system.with_a_sensor
subcomponents
  the_sensor : device a_sensor
    {our_properties::pressure_max => 1020 hPa;};
end a_system.with_a_sensor;
```

Raffinement de la sémantique d'exécution - 1

- L'élément central d'une description AADL est le thread (tâche)
- La sémantique d'exécution 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
  - *Scheduling_Protocol* précise la politique d'ordonnancement associé à un processeur (rappel: le processeur AADL n'est pas fait que de matériel).
  - *Compute_execution_time* représente le temps d'exécution d'un thread ou d'un sous-programme.
  - *Source_Stack_Size* définit la taille de pile allouée au thread.

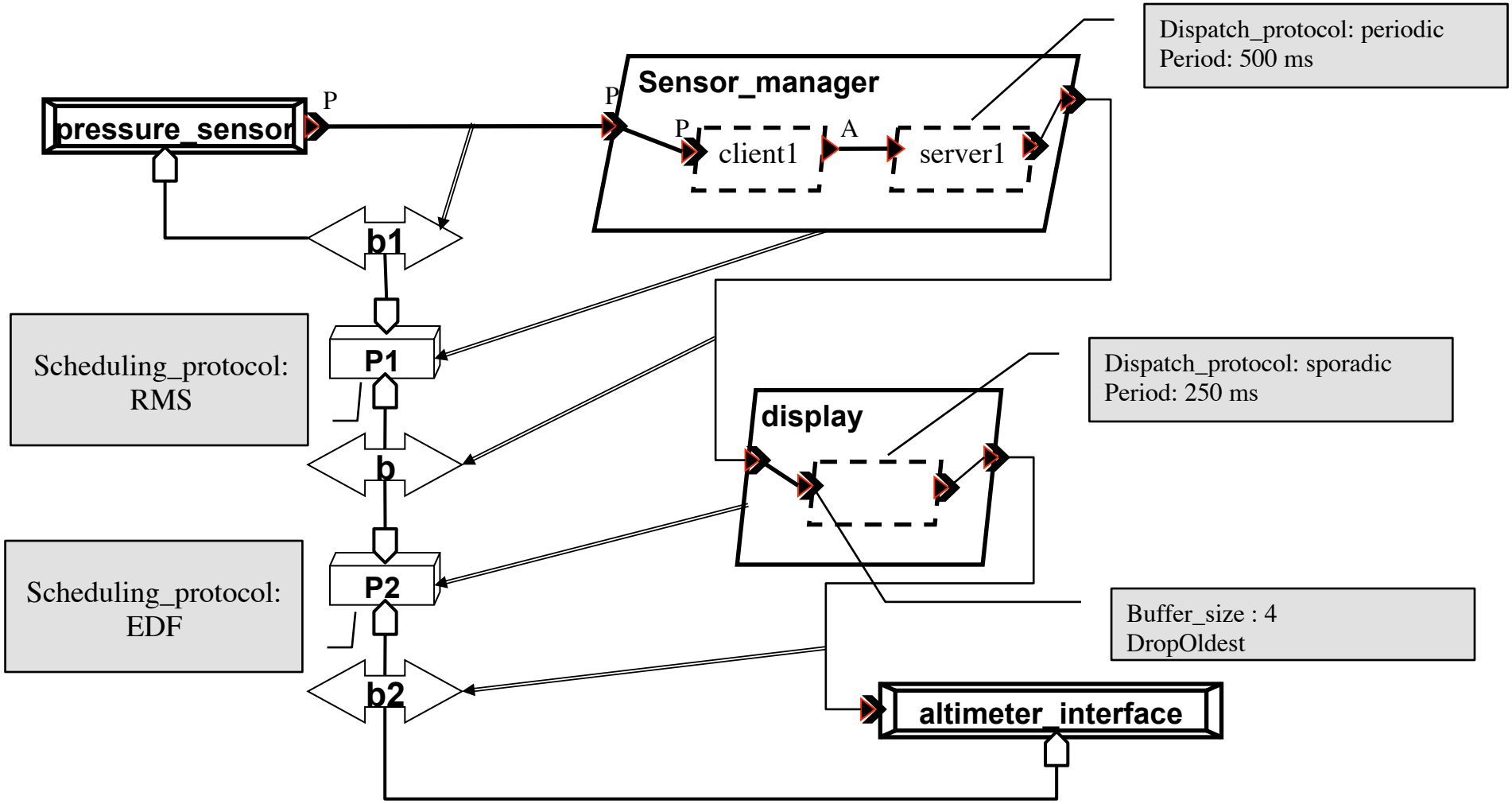
Raffinement de la sémantique d'exécution - 2

- Contrairement aux data ports, les event ports peuvent être mis en file d'attente
  - L'information envoyée sur un port data peut écraser la précédente
  - La longueur de la file peut être spécifié par l'intermédiaire d'une *propriété* AADL
  - Si la file d'attente est pleine, une des politique suivantes peut être adoptée:
    - DropOldest: On supprime le message le plus ancien
    - DropNewest: On supprime le message le plus récent
  - La politique de dépilage est spécifiée par la propriété *Dequeue_Protocol*:
    - *OneItem* (un seul élément est dépilé à la fois)
    - *AllItems* (tous les éléments sont systématiquement dépilés)
    - *MultipleItems* (la quantité dépilée est spécifié par un nombre entier)
- Modélisation de la politique d'accès aux variables partagées
  - La propriété *Concurrency_Control_Protocol* specifie la politique de protection de la donnée (RWLock, PIP, PCP, ICPP, etc…).

Spécification du déploiement

- Des composants logiciels sur la plate-forme d'exécution
  - *Actual_Processor_Binding* **references <**processor component> **applies to** <Processes or threads>
  - *Actual_Memory_Binding* **references** <memory component> **applies to <**threads, processes, data, or data port>
- Des connections sur les bus
  - *Actual_Connection_Binding* **reference** <bus component> **applies to** <connection>

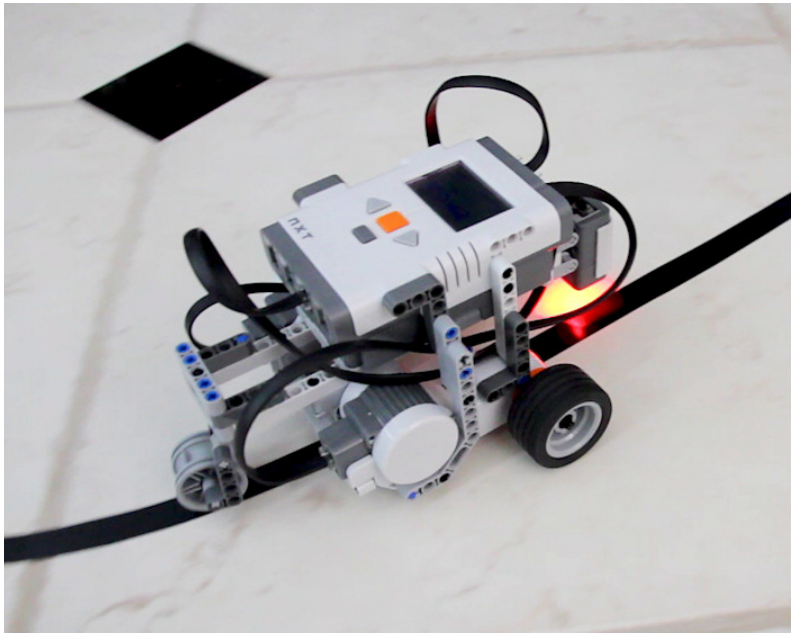
Exemple complet: altimètre



Exemple: intégration de code “legacy”

Robot suiveur de ligne

```
#define KP 60
#define KI 15
int integral=0;
int lastColor = 1;

/**
 * Computes the angle the robot has to turn in
 * order to follow the line.
 */
void computePID(int in_light, int colorToFollow, int *out_angle) {
  char error;
  if(in_light == 99)
    error = 0;
  else if(in_light ==  colorToFollow) {
      error = 20;
      if(lastColor=1)
          integral=0;
      lastColor=0;
    }
  else {
      if(lastColor=0)
          integral=0;
      error = -20;
      lastColor=1;
    }
  integral = integral + error; //  multiply by period
  if(integral>90)
    integral = 90;
  if(integral<-120)
      integral=-120;
  *out_angle = (KP * error + KI*integral)/100;
}
```



Représentation des données en AADL

- La catégorie de composants AADL utilisé pour décrire une donnée est de la catégorie « data », qui représente soit:
  - Un type de donnée simple,
  - Une structure de donnée,
  - Une instance de donnée en mémoire.

- Des propriétés permettent de référencer des types de donnée existant, ainsi que les propriétés non-fonctionnelles (occupation mémoire, unités de la donnée, etc.)

- Ces propriétés sont fournis dans un ensemble de propriétés standardisé (voir extrait ci-après).

Exemple de propriétés pour les types de donné

```
property set Data_Model is
…
 Data_Representation : enumeration
     (Array, Boolean, Character, Enum, Float,
      Fixed, Integer, String, Struct, Union)
      applies to ( data, feature );
   --  The Data_Representation property may be used to specify the
   --  representation of simple or composite data types within the
   --  programming language source code.
 Enumerators : list of aadlstring applies to ( data, feature );
   --  The Enumerators provides the list of enumeration litterals
   --  attached to an enumeration data component.
…
end Data_Model;
```

Données utilisé par la fonction de suivi de ligne

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


data Color
properties
  Data_Model::Data_Representation => Enum;
  Data_Model::Enumerators => ("NXT_COLOR_BLACK", "NXT_COLOR_RED",
                              "NXT_COLOR_BLUE");
  Source_name => "COLOR_TYPE";
  Source_Text => ("ecrobot_interface.h");
end Color;
```

Représentation du code en AADL

- La catégorie de composants AADL utilisé pour décrire un code source sont de la catégorie « sous-programmes », qui représente une fonction (C, Ada, Java, etc.).

- Un sous-programme possède des paramètres avec une direction (in, out, in-out) et un type de donnée

- Des propriétés permettent de référencer le code source existant, ainsi que les propriétés non-fonctionnelles

Code de la fonction de suivi de ligne

```
subprogram computePID
features
  currentColor : in parameter Int;
  colorToFollow: in parameter Color;
  angle : out parameter Int;
properties
  Source_language => (C);
  Source_text => ("in_robot.h", "in_robot.c");
  Source_name => "computePID";
end computePID;
```

Exemple: analyse d'ordonnançabilité (1/2)

Rappel des propriétés d'ordonnancement

- Processor:
  - ⌘ Scheduling_Protocol
- Thread:
  - ⌘ Dispatch_protocol
  - ⌘ Period
  - ⌘ Execution_Time
  - ⌘ Deadline

Exemple (1/2)

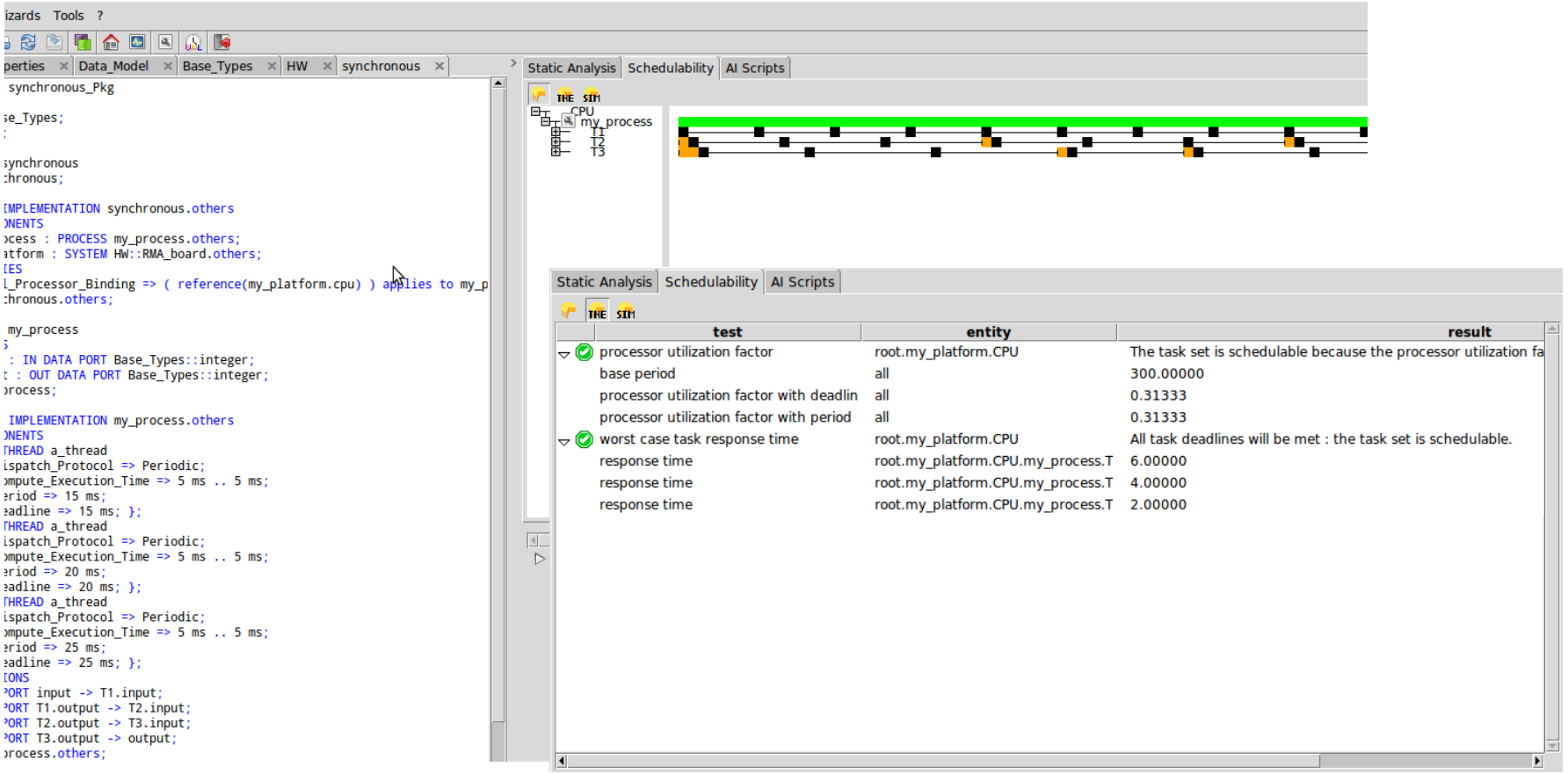
```
PACKAGE synchronous_Pkg
PUBLIC
WITH Base_Types;

SYSTEM synchronous
END synchronous;

SYSTEM IMPLEMENTATION synchronous.others
SUBCOMPONENTS
my_platform : PROCESSOR CPU;
my_process : PROCESS my_process.others;
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
FEATURES
input : IN DATA PORT Base_Types::integer;
output : OUT DATA PORT
Base_Types::integer;
END my_process;
```

Exemple (2/2)

```
PROCESS IMPLEMENTATION my_process.others
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; };
CONNECTIONS
C0 : PORT input -> T1.input;
C1 : PORT T1.output -> T2.input;
C2 : PORT T2.output -> T3.input;
C3 : PORT T3.output -> output;
END my_process.others;

THREAD a_thread
FEATURES
input : IN DATA PORT Base_Types::integer;
output : OUT DATA PORT
Base_Types::integer;
ANNEX Behavior_Specification {**
VARIABLES x : Base_Types::integer;
STATES s : INITIAL COMPLETE FINAL STATE;
TRANSITIONS t : s -[ ON DISPATCH ]-> s
{ x := input + 1; output := x };
**};
END a_thread;

END synchronous_Pkg;
```

Exploitation avec AADL Inspector



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;
```

Représenter un accès à une ressource partagée

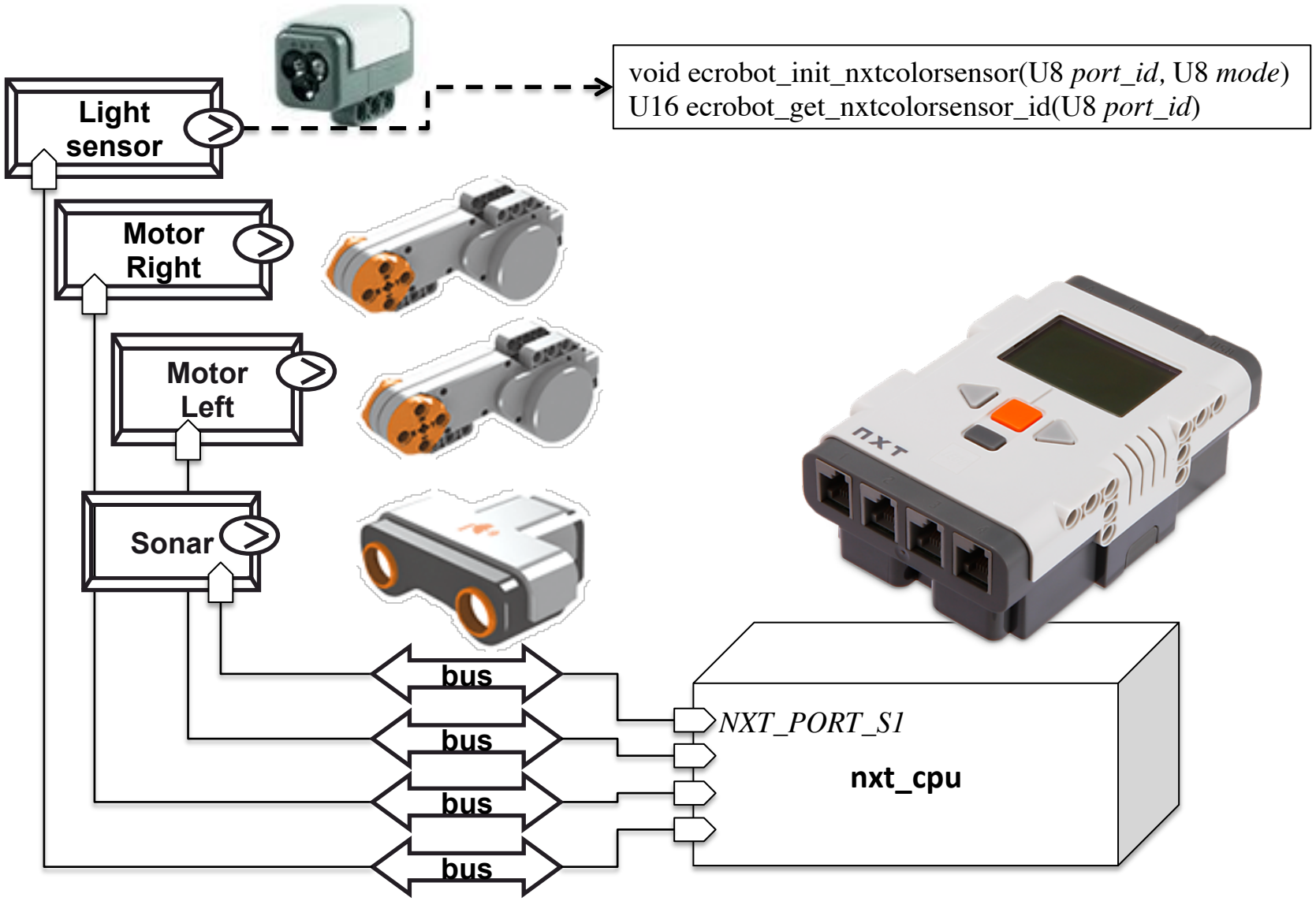
```
data d
end d;

subprogram spg
features
  S : requires data access d;
end spg;

thread t
features
  S : requires data access d;
end t;
```

```
thread implementation t.example
calls {
  call1 : subprogram d.spg;
};
connection
  cnx : data access S -> call1.S;
end t.example;
```

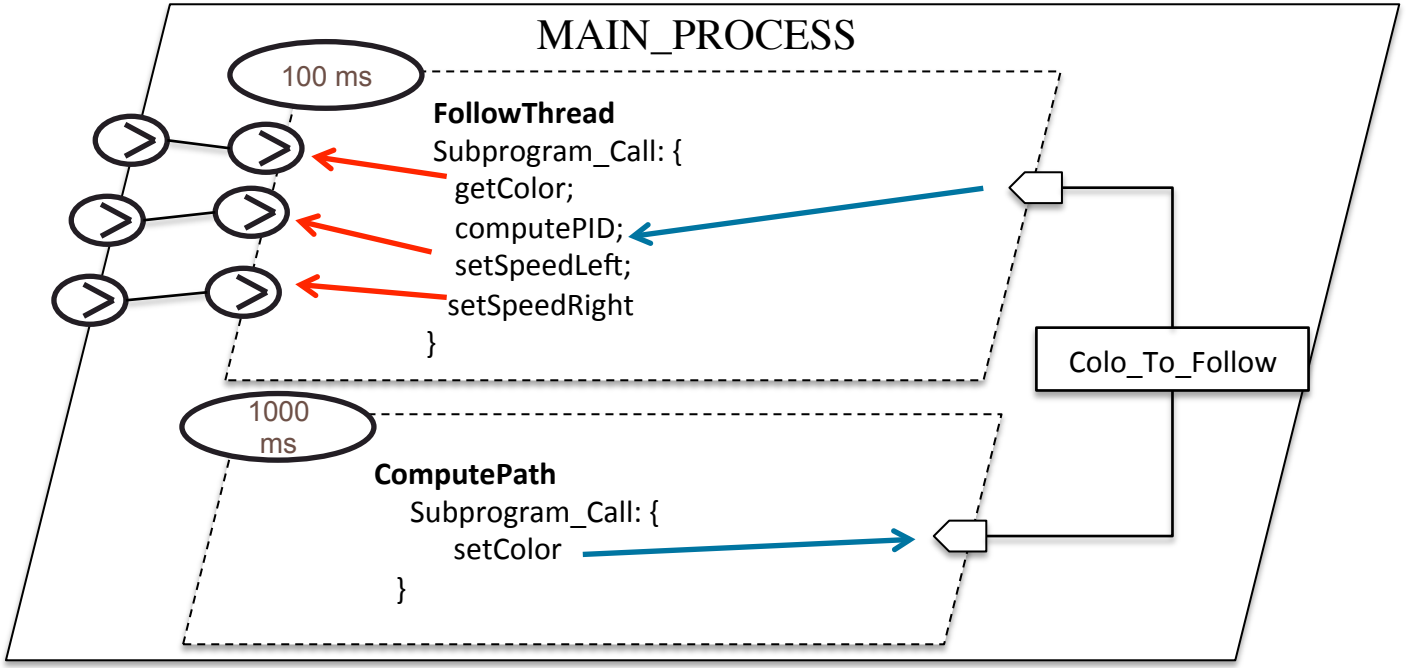
Continuons l'intégration de code legacy dans le robot Lego

- Nous avons représenté le code legacy, (computePID) en AADL
- Pour intégrer ce code, il va falloir
  - L'activer → utilisation de thread AADL
  - Lui fournir les valeurs d'entré et utiliser ses sorties
    - `int in_light, → fourni par la plate-forme d'exécution`
    - `int colorToFollow, → fourni par une autre tâche`
    - `int *out_angle → à transmettre à la plate-forme d'exécution`

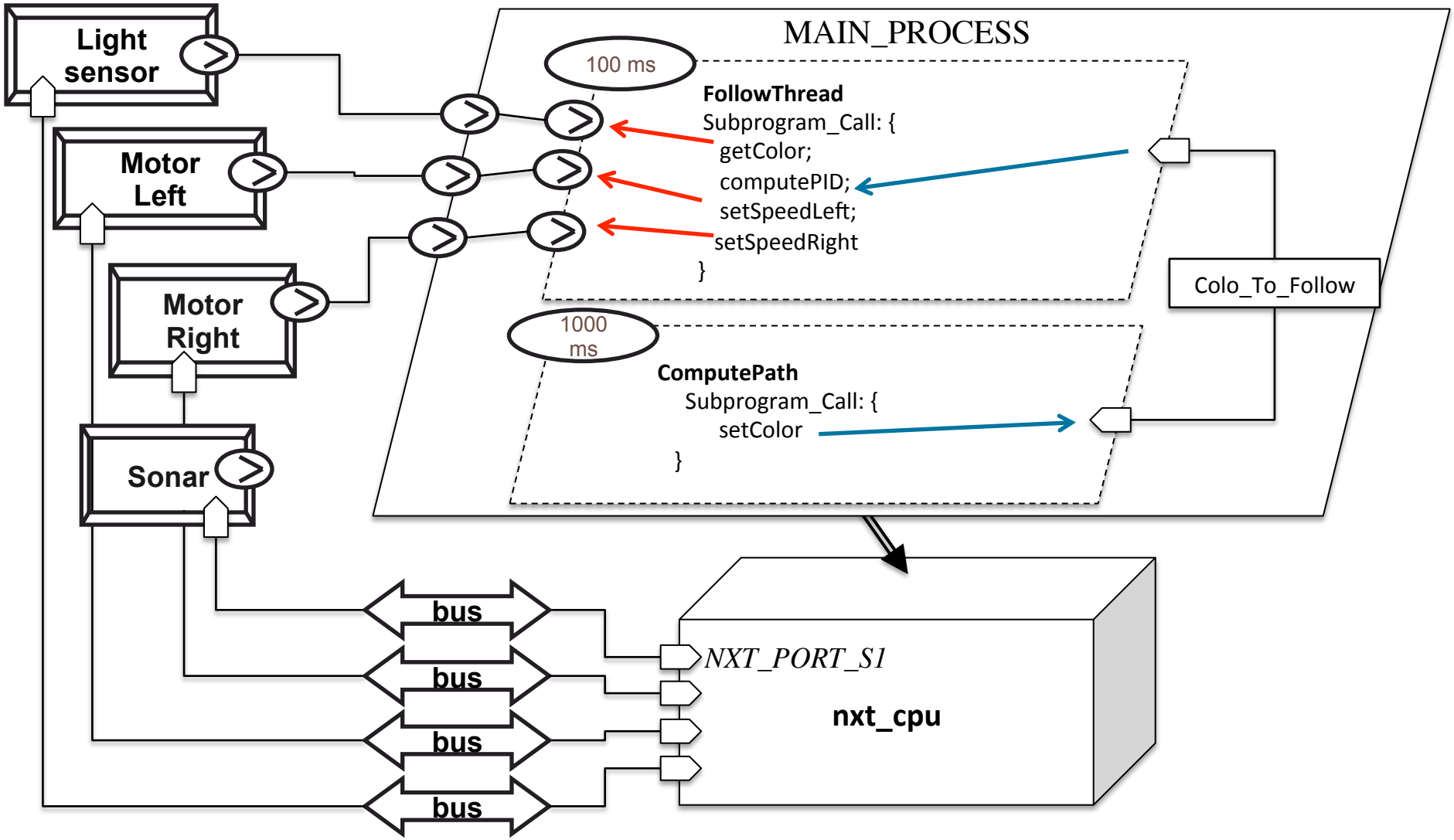
Représentation du support d'exécution du robot en AADL



Intégration du code dans une architecture temps-réel



Représentation de l'architecture intégrée du robot



Représentation des sections critiques

- Données partagées
  - ⌘ Si protégée, il faut prendre en compte les interférences dues aux sections critiques (temps de blocage).
  - ⌘ Il faut donc représenter plus finement le comportement du thread
- Annexe comportementale (voir slides suivants)

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 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

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 message contenus dans le buffer associé à p;
  - Le contenu de la valeur retourné dépend du protocole associé au port (propriété *Dequeue_Protocol*)
- p'count et p'fresh pour connaitre l'état d'un buffer correspondant à un port p
  - p'count retourne le nombre d'éléments dans le buffer
  - p'fresh retourne 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;
```

Exemple: analyse d'ordonnançabilité (2/2)

Code AADL (1/3)

```
PACKAGE shared_data_Pkg
PUBLIC

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

SYSTEM shared_data
END shared_data;

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

```
PROCESS my_process
END my_process;

PROCESS IMPLEMENTATION my_process.others
SUBCOMPONENTS
T1 : THREAD T.i1;
D1 : DATA D

    { Concurrency_Control_Protocol =>
Priority_Ceiling; };
D2 : DATA D

    { Concurrency_Control_Protocol =>
Priority_Ceiling; };
T2 : THREAD T.i2;
CONNECTIONS
C1 : DATA ACCESS D1 -> T1.D1;
C2 : DATA ACCESS D2 -> T1.D2;
C3 : DATA ACCESS D1 -> T2.D1;
C4 : DATA ACCESS D2 -> T2.D2;
END my_process.others;
```

Code AADL (2/3)

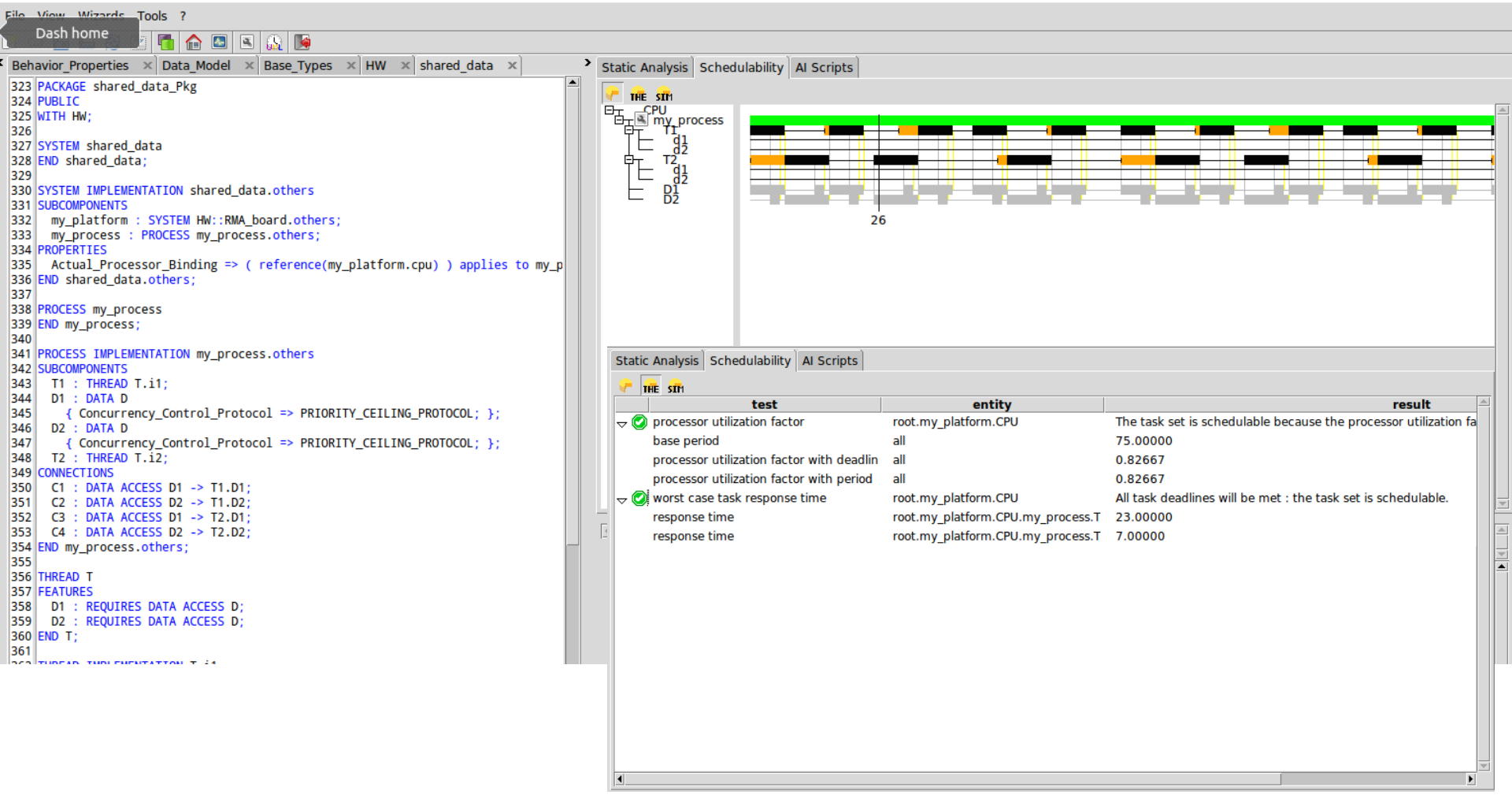
```
THREAD T
FEATURES
D1 : REQUIRES DATA ACCESS D;
D2 : REQUIRES DATA ACCESS D;
END T;


DATA D
END D;
```

```
THREAD IMPLEMENTATION T.i1
PROPERTIES
Dispatch_Protocol => Periodic;
Compute_Execution_Time => 5ms..5ms;
Period => 15 ms;
ANNEX Behavior_Specification {**
States
    s : initial complete final state;
Transitions
    t : s -[on dispatch]-> s {
      D1 !<;
      computation(3 ms);
      D2 !<;
      D2 !>;
      D1 !>
};
**};
END T.i1;
```

Code AADL (3/3)

```
THREAD IMPLEMENTATION T.i2
PROPERTIES
Dispatch_Protocol => Periodic;
Compute_Execution_Time => 5ms..5ms;
Period => 25 ms;
ANNEX Behavior_Specification {**
States
    s : initial complete final state;
Transitions
    t : s -[on dispatch]-> s {
      D2 !<;
      computation(5 ms);
      D1 !<;
      D1 !>;
      D2 !>
};
**};
END T.i2;

END shared_data_Pkg;
```

Exploitation avec AADL Inspector



Plus abstraits: Les flots (1)

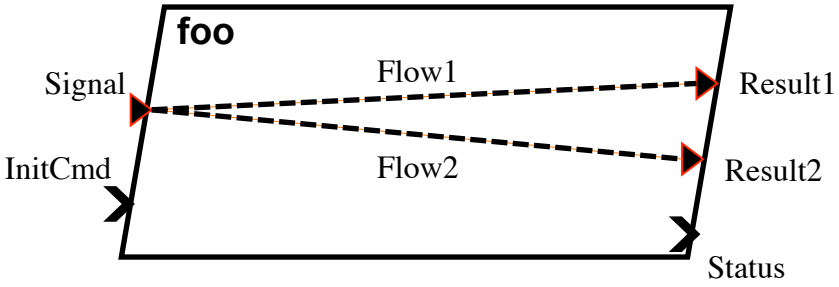
- Les flots permettent de matérialiser les chemins à travers les éléments d'une description. Ils suivent plus ou moins les connexions.
  - ne modélisent rien de concret
  - une forme de redondance pour pouvoir analyser plus facilement les flots de données et y associer des propriétés
- on peut décrire :
  - le début d'un flot (`flow source`)
  - le milieu d'un flot (`flow path`)
  - la fin d'un flot (`flow sink`)
  - un flot complet (`end to end flow`)

Les flots (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

```
process foo
features
  InitCmd : in  event      port;
  Signal  : in        data port signal_data;
  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 sink   InitCmd;
  Flow4 : flow source Status;
end foo;
```



Exemple de flow implementation

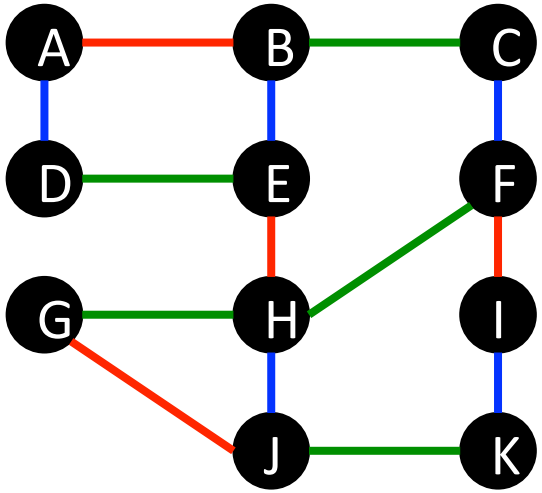
```
process implementation foo.basic
subcomponents
A: thread bar.basic;
-- bar has a flow path fs1 from port p1 to p2
-- bar has a flow source fs2 to p3
C: thread baz.basic;
B: thread baz.basic;
-- baz has a flow path fs1 from port p1 to p2
-- baz has a flow sink fsink in port reset
connections
Conn1        : data  port signal  -> A.p1;
Conn2        : data  port A.p2    -> B.p1;
Conn3        : data  port B.p2    -> result1;
Conn4        : data  port A.p2    -> C.p1;
Conn5        : data  port C.p2    -> result2;
Conn6        : data  port A.p3    -> status;
connToThread : event port initcmd -> C.reset;
flows
Flow1: flow path signal -> conn1 -> A.fs1 -> conn2 -> B.fs1 -> conn3 -> result2;
Flow2: flow path signal -> conn1 -> A.fs1 -> conn4 -> C.fs1 -> conn5 -> result2;
Flow3: flow sink initcmd -> connToThread -> C.fsink;
-- a flow source may start in a subcomponent,
-- i.e., the first named element is a flow source
Flow4: flow source A.fs2 -> connect6 -> status;
end foo.basic;
```

Exemple: Gestion des modes d'un robot mobile

Modes de fonctionnement

- Définition: un mode de fonctionnement représente un état du système dans lequel celui-ci fournit un sous-ensemble de ses fonctionnalité.

- Exemple: Aller de C à J dans la carte (initialement le robot est sur un point noir)
  - ⌘ Calculer un chemin (séquence de couleurs)
  - ⌘ Chercher la prochaine ligne à suivre
  - ⌘ Détecter l'atteinte de la ligne à suivre
  - ⌘ Suivre une ligne de couleur
  - ⌘ Détecter l'atteinte d'un point noir
  - ⌘ Détecter des obstacles sur la trajectoire
  - ⌘ Arrêter le robot en cas d'obstacle



Fonctionnalités par mode

- Modes
  1. Calculer un chemin
  2. Chercher la prochaine ligne
  3. Suivre une ligne
- Fonctionnalités par mode
  1. Calculer un chemin (séquence de couleurs).
  2. Chercher la prochaine ligne à suivre, Détecter l'atteinte de la ligne, Détecter des obstacles sur la trajectoire, Arrêter le robot en cas d'obstacle.
  3. Suivre une ligne de couleur, Détecter l'atteinte d'un point noir, Détecter des obstacles sur la trajectoire, Arrêter le robot en cas d'obstacle.

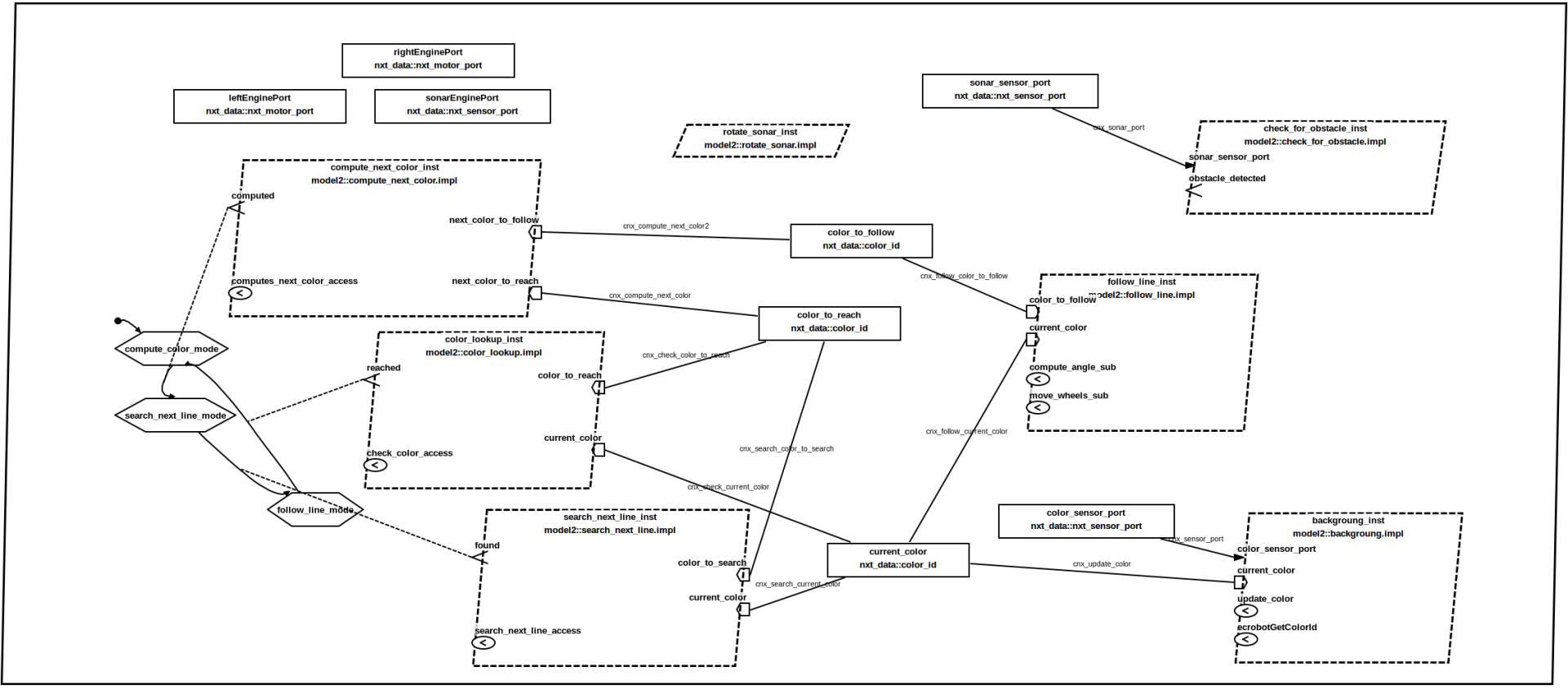
Les modes en AADL

- Les modes permettent de modéliser la reconfiguration du système en fonction d'événements
  - définis au niveau des composants
  - seuls les événements (event ports) peuvent déclencher un changement de mode
  - certains sous-composants, connexions, etc. ne sont activés que dans certains modes
  - la définition des propriété peut dépendre du mode courant
- Les changements de modes permettent de représenter des architectures (pseudo) dynamiques
  - évolution (dynamique) de la configuration de l'architecture
  - ensemble déterminé (statiquement) de configurations possibles

Exemple de modes

```
thread execution_thread
end execution_thread;

process a_process
features
  multi_thread : in event port;
  mono_thread : in event port;
end a_process;

process implementation a_process.configurable
subcomponents
  thread_1 : thread execution_thread;
  thread_2 : thread execution_thread in modes (multitask);
modes
  monotask : initial mode;
  multitask : mode;
  monotask -[ multi_thread ]-> multitask;
  multitask -[ mono_thread ]-> monotask;
end a_process.configurable;
```

Mise en œuvre avec le robot



Architecture par mode

- Mode compute path: 1 tâche de calcul de la séquence de lignes à suivre.
- Mode follow line: 4 tâches
  - suivi de ligne (avec computePID)
  - détection d'atteinte d'un point noir (changement de mode)
  - détection d'osbtacle
  - MàJ de la couleur (tâche background requise par la plate-forme d'exécution)
- Mode search next line: 4 tâches
  - suivi de point noir
  - détection d'atteinte de la prochaine ligne (changement de mode)
  - détection d'osbtacle
  - MàJ de la couleur (tâche background requise par la plate-forme d'exécution)
- L'architecture instanciée dans chaque mode est décrite en utilisant « in modes »:

Transitions de mode

- Compute path → Search next line
  - ⌘ Une fois le chemin calculé, on cherche la première ligne à suivre
- Search next line → Mode follow line
  - ⌘ Une fois la ligne atteinte, on la suit
- Mode follow line→ Search next line
  - ⌘ Une fois le point noir atteint, on cherche la prochaine ligne

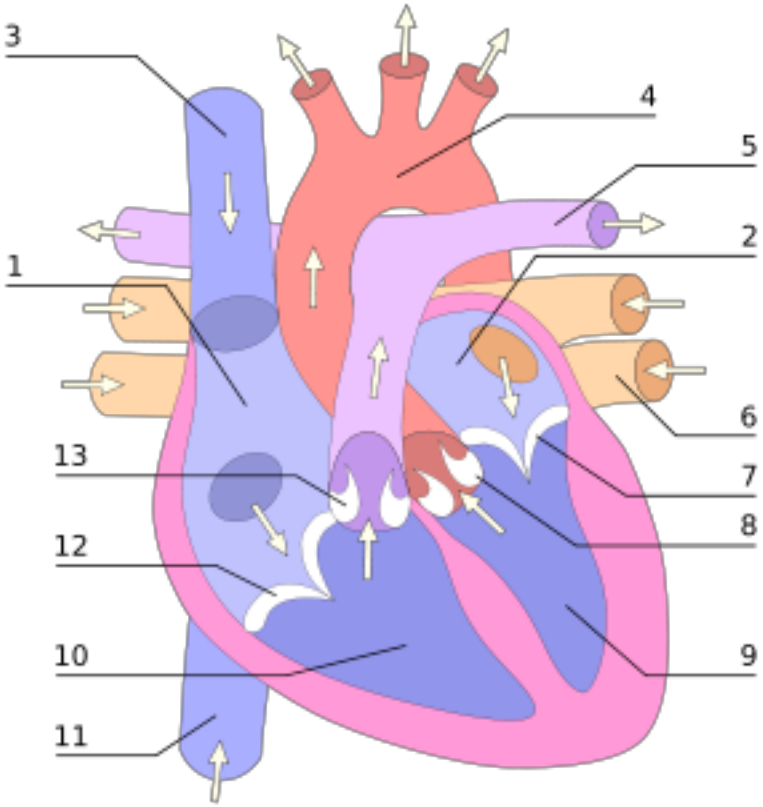
- L'architecture instanciée dans chaque mode est décrite en utilisant « in modes »:

Code AADL

```
process implementation proc.impl
    subcomponents
        backgroung_inst : thread backgroung.impl in modes (follow_line_mode,
                                                           search_next_line_mode);
        follow_line_inst : thread follow_line.impl in modes  (follow_line_mode);
        color_lookup_inst : thread color_lookup.impl in modes   (follow_line_mode);
    …
    modes
        follow_line_mode : mode;
        compute_color_mode : initial mode;
        search_next_line_mode : mode;
        follow_line_mode -[color_lookup_inst.reached]-> compute_color_mode;
        compute_color_mode -[compute_next_color_inst.computed]->
                              search_next_line_mode;
        search_next_line_mode -[search_next_line_inst.found]-> follow_line_mode;
    …
end proc.impl;
```

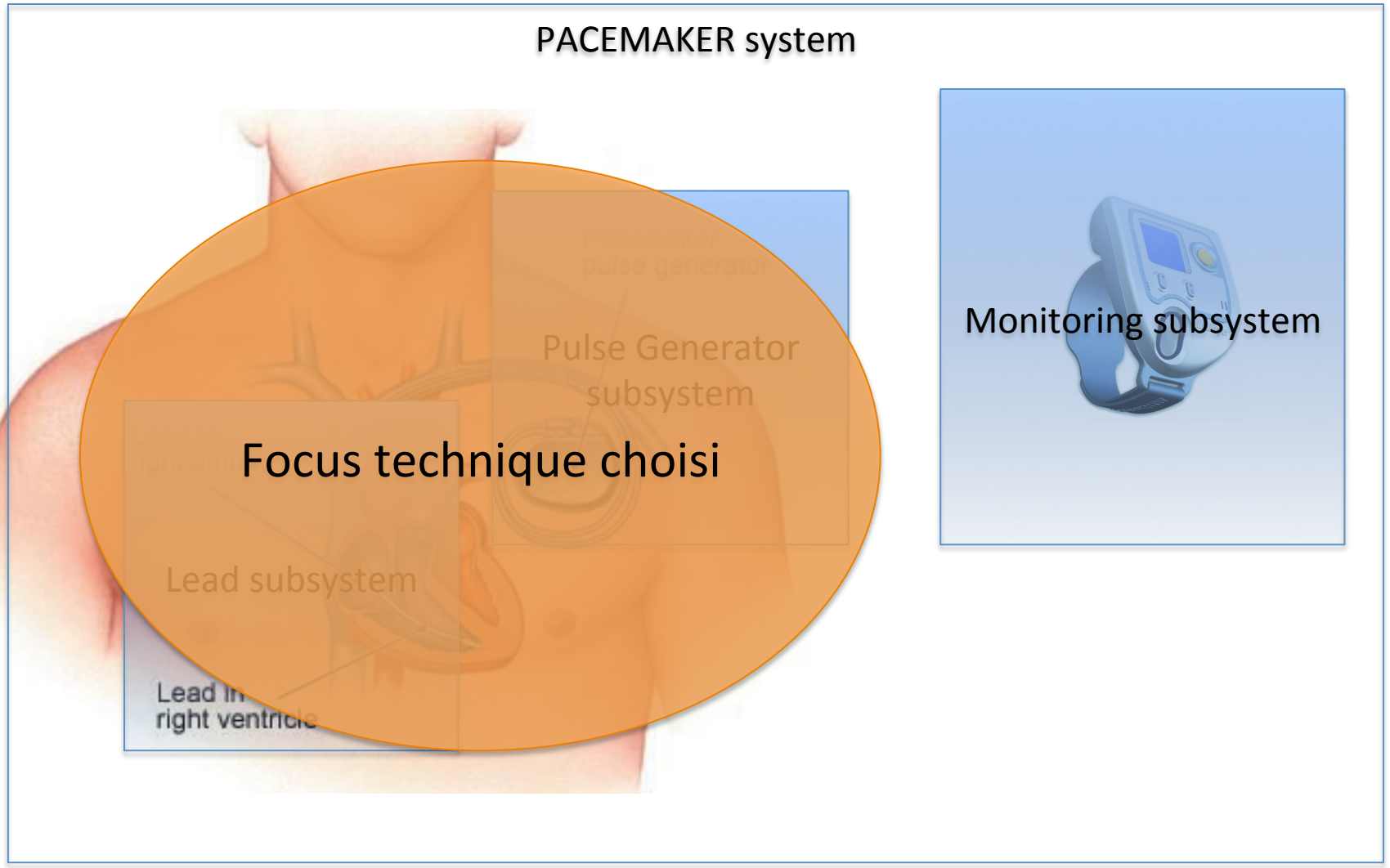
C’as d’étude: spécification, conception, et vérification d’un pacemaker

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 tricuspide (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

On va s'intéresser au comportement du pacemaker dans les modes **DOOR et AAI**

Exigences temporelles

Parameter	AAT	VVT	AOO	AAI	VOO	VVI	VDD	DOO	DDI	DDD	AOOR	AAIR	VOOR	VVIR	VDDR	DOOR	DDIR	DDDR
Lower Rate Limit	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
Maximum Sensor Rate											X	X	X	X	X	X	X	X
Fixed AV Delay							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
Ventricular Amplitude		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
Ventricular Pulse Width		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
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
Reaction Time											X	X	X	X	X	X	X	X
Response Factor											X	X	X	X	X	X	X	X
Recovery Time											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

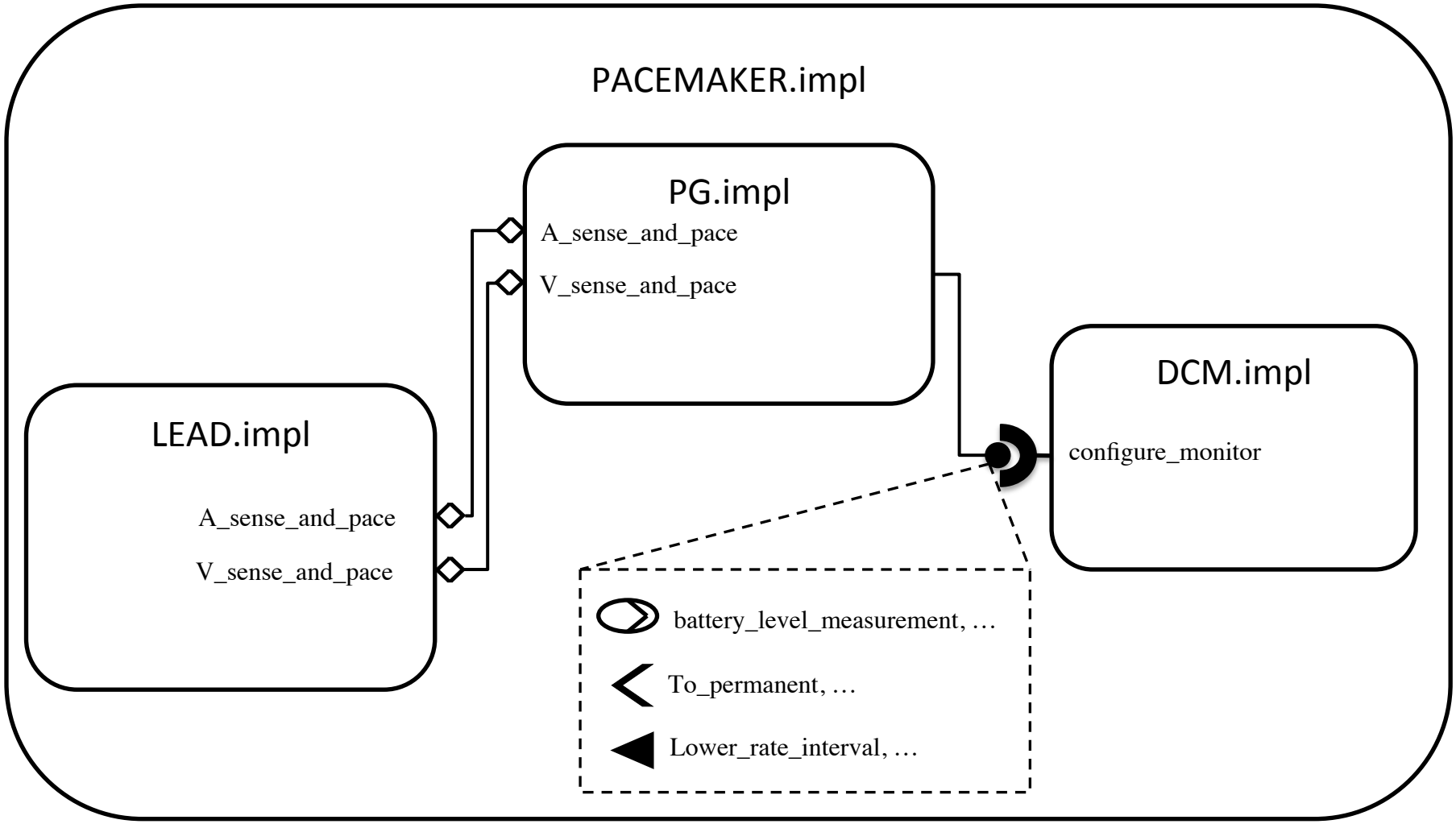
AV: durée séparant un battement dans l'oreillette d'un battement dans le ventricule

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

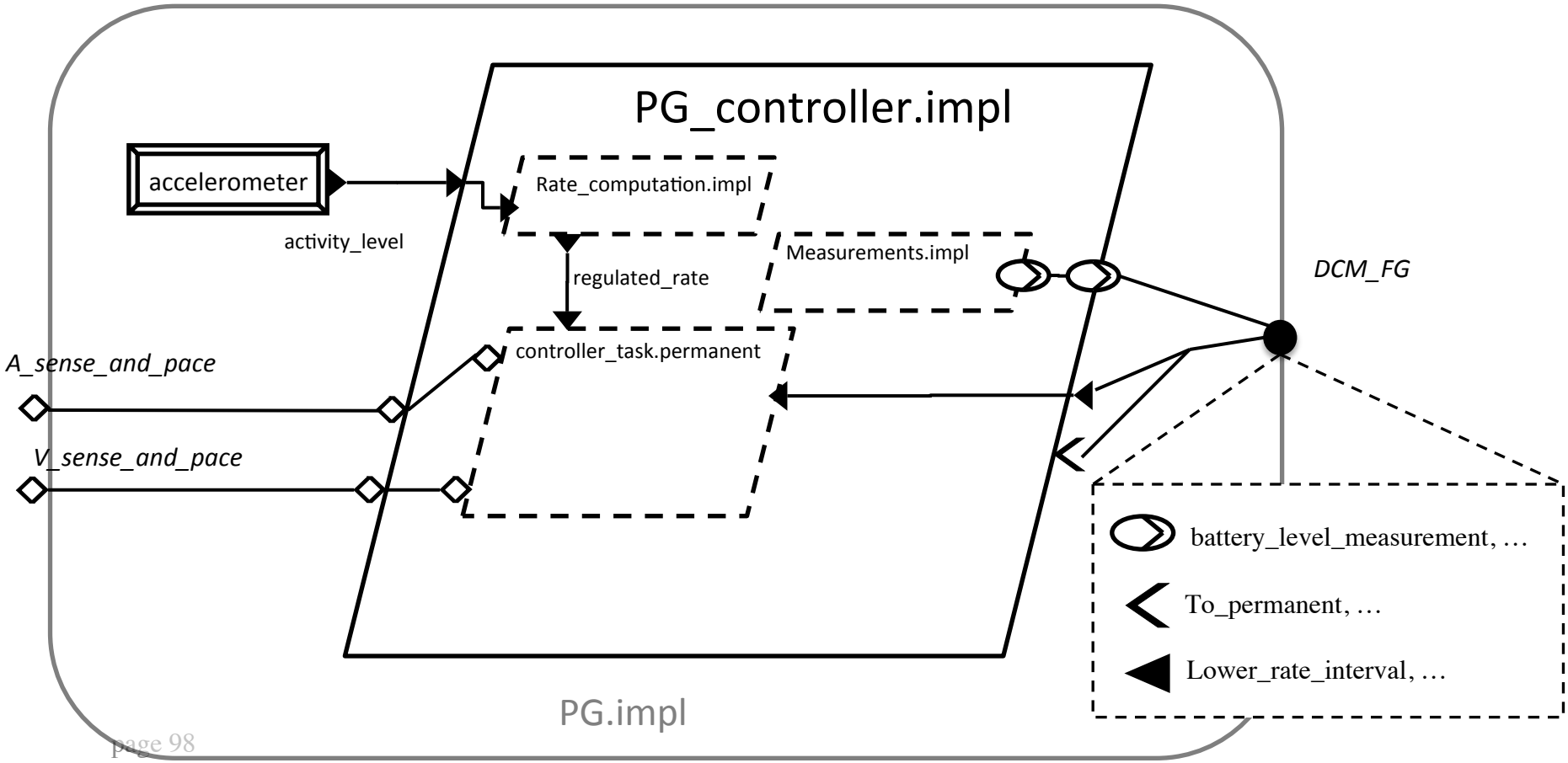


Identification des composants logiciels et de leurs interfaces

- 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

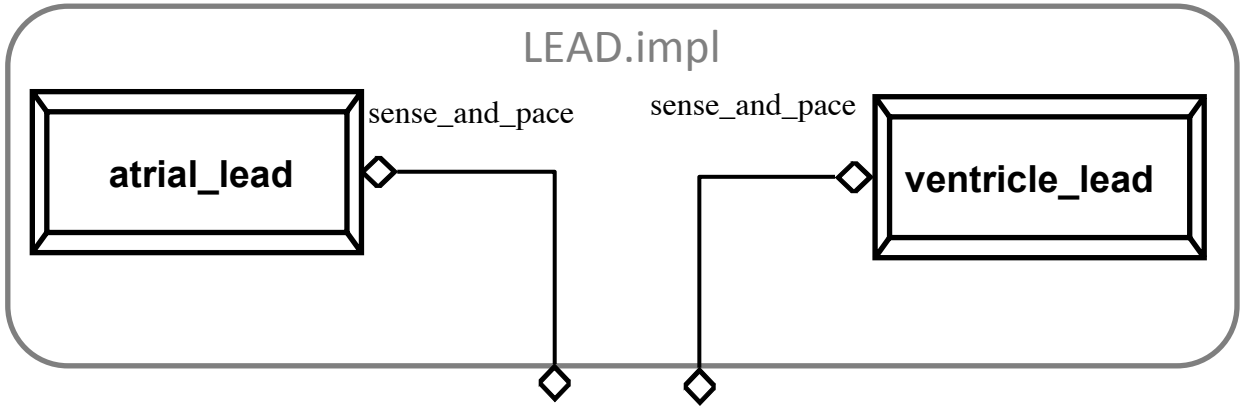
- Ensembles de connections triviales:
  - ⌘ Périphériques de capture → entrées du processus de contrôle
  - ⌘ Sorties du processus de contrôle → périphériques de stimulation

PG: processus de controle



Composants du sous-système LEAD

- **Leads**
  - Capteurs (ventriculaire/atriale) des battements naturels
  - Stimulation (ventriculaire/atriale) du coeur

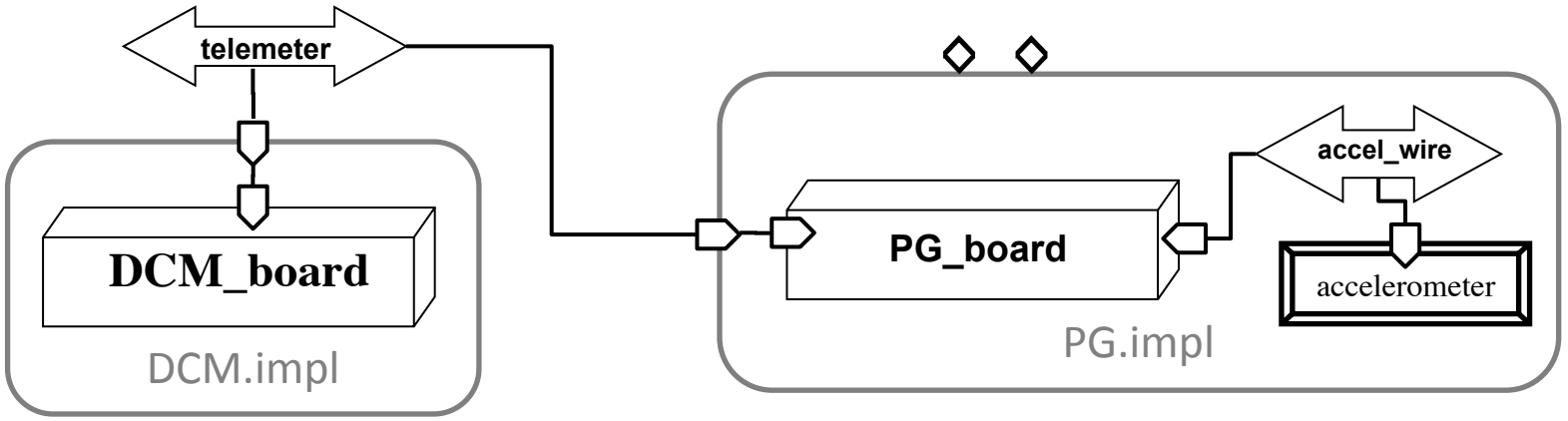


Modèle AADL: Lead system

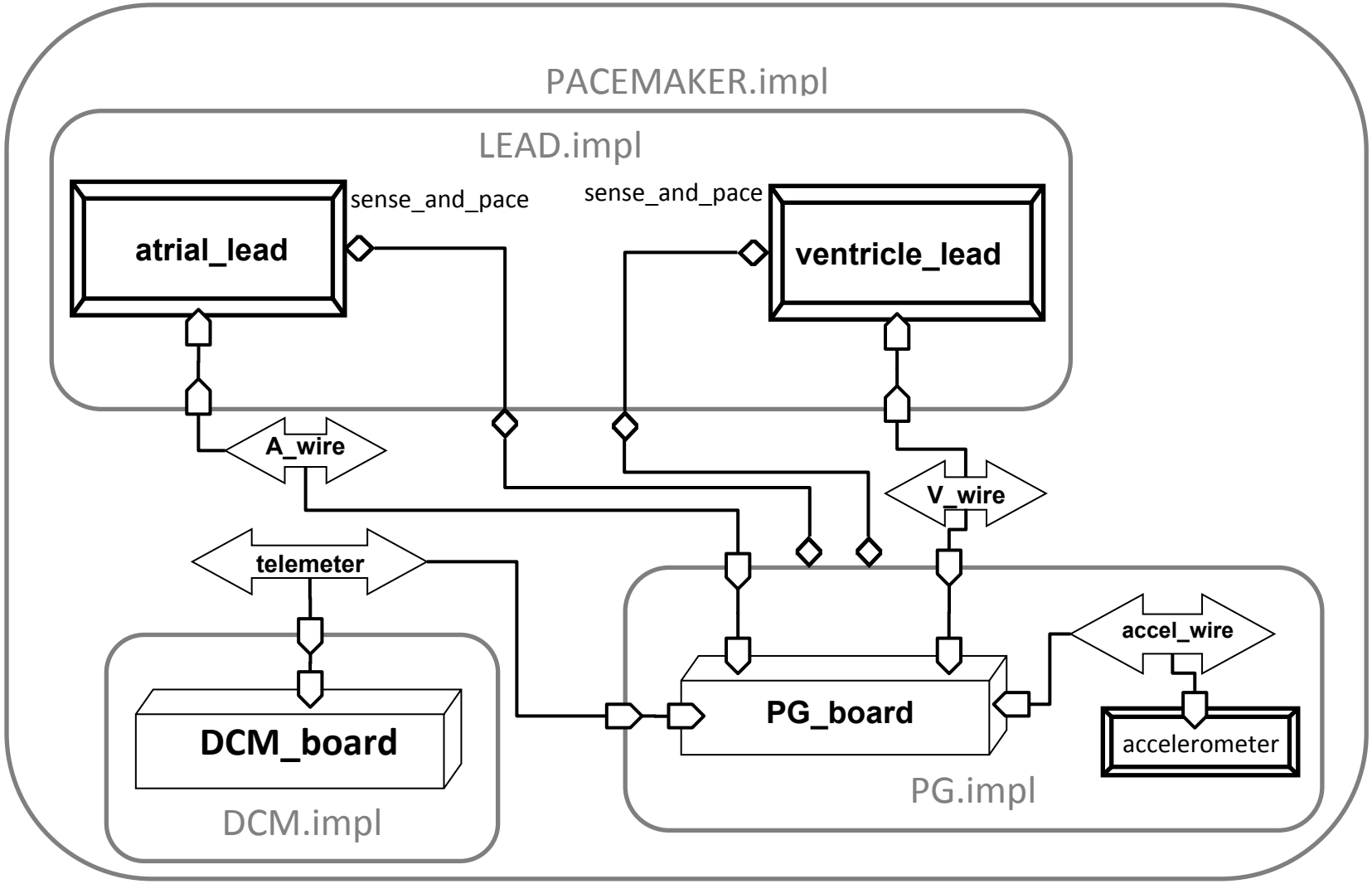
```
device compartiment_lead
  features
    sense_and_pace: in out event port;
end compartiment_lead;

system implementation LEAD.impl
  subcomponents
    atrial_lead: device compartiment_lead;
    ventricle_lead: device compartiment_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'execution 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)

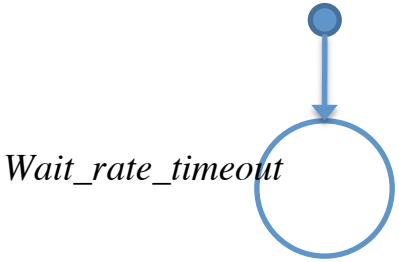
Ordonnancement des tâches

- *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'oronnancement est RMS
  - Ex: Period ne peut prendre q'une valeur entière constante

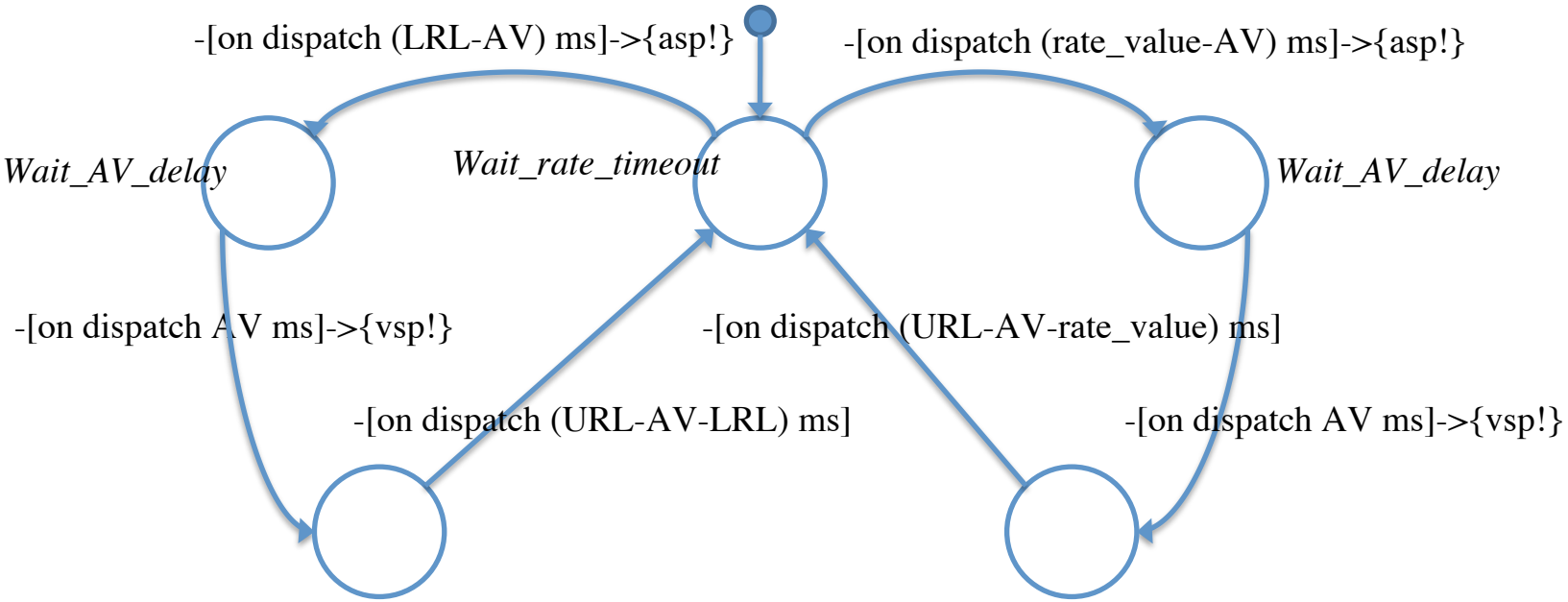
Comportement de la tâche de contrôle dans le mode DOOR

- Contraintes temporelles:
  - ⌘ LRL, AV, URL, et rate_value



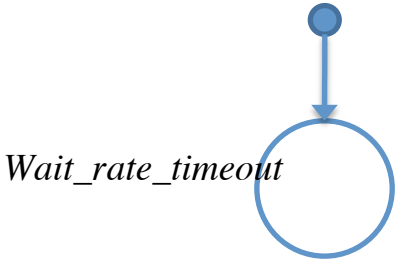
Comportement de la tâche de contrôle dans le mode DOOR

- Contraintes temporelles:
  - ⌘ LRL, AV, URL, et rate_value



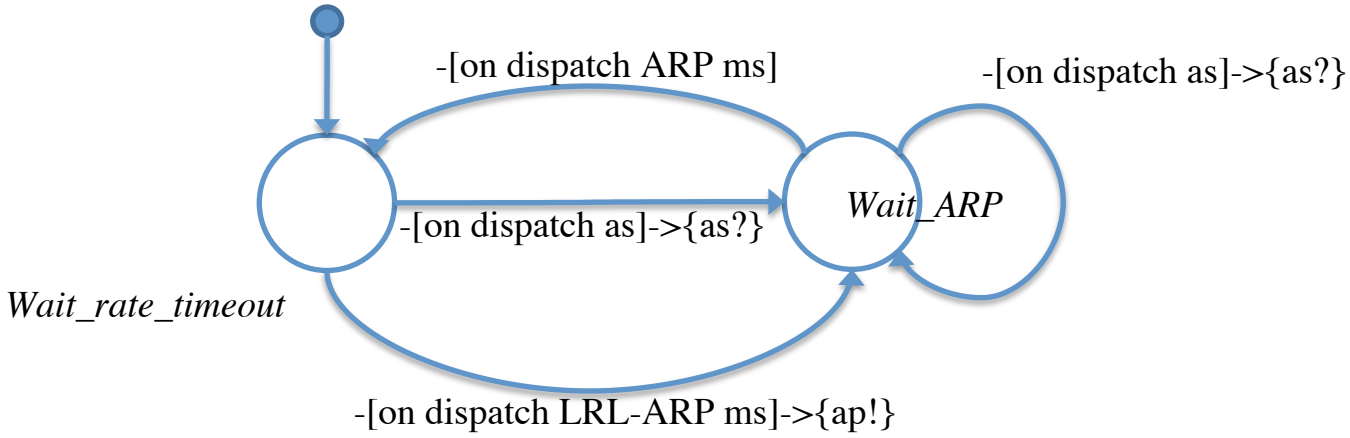
Comportement de la tâche de contrôle dans le mode AAI

- 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 comportemetale

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

Défauts de la conception: partie comportemetale

- La solution proposée permet-elle d'assurer le respect des exigences temporelles?
- Quelque 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?

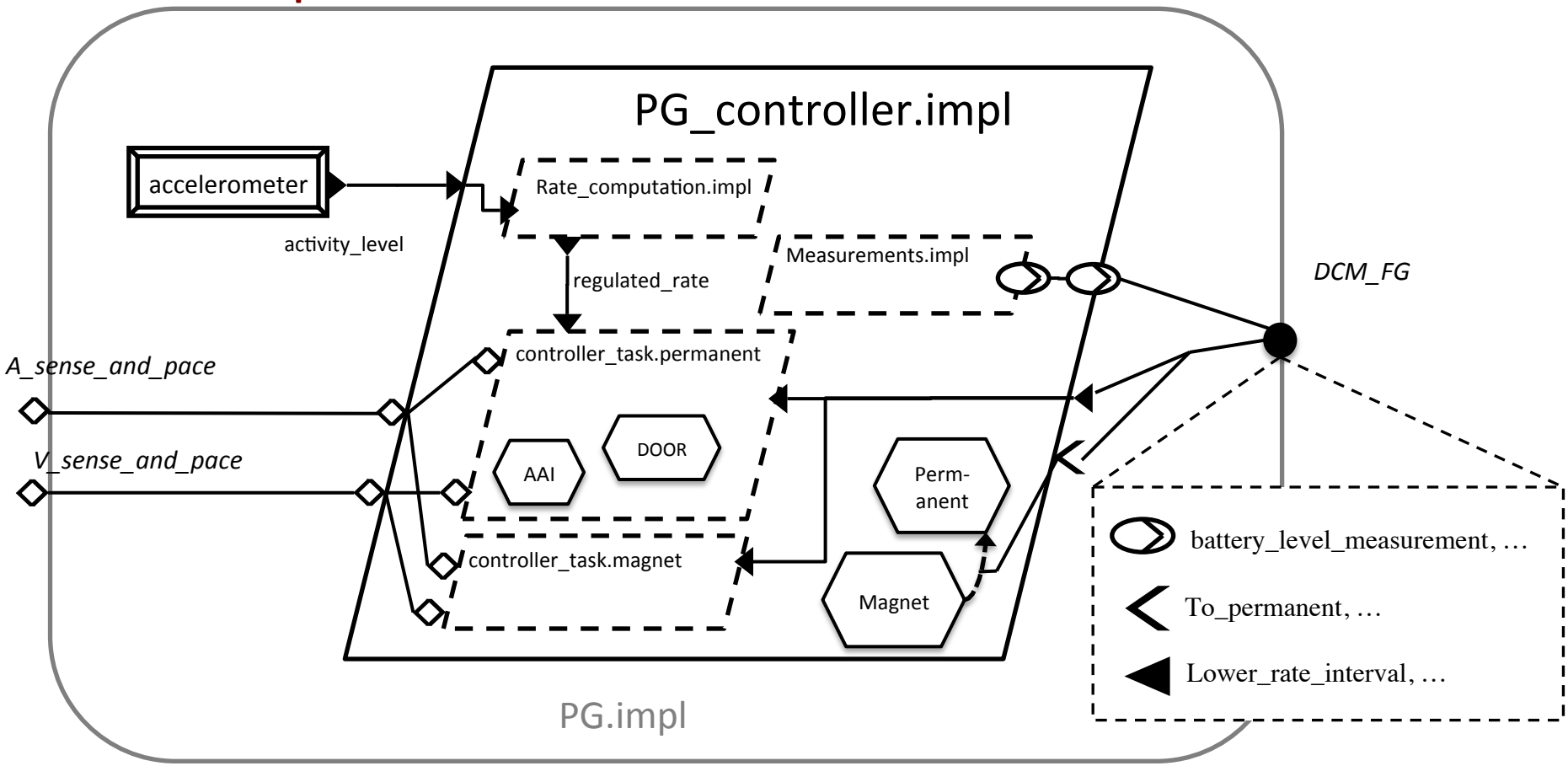
Modes de fonctionnement

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

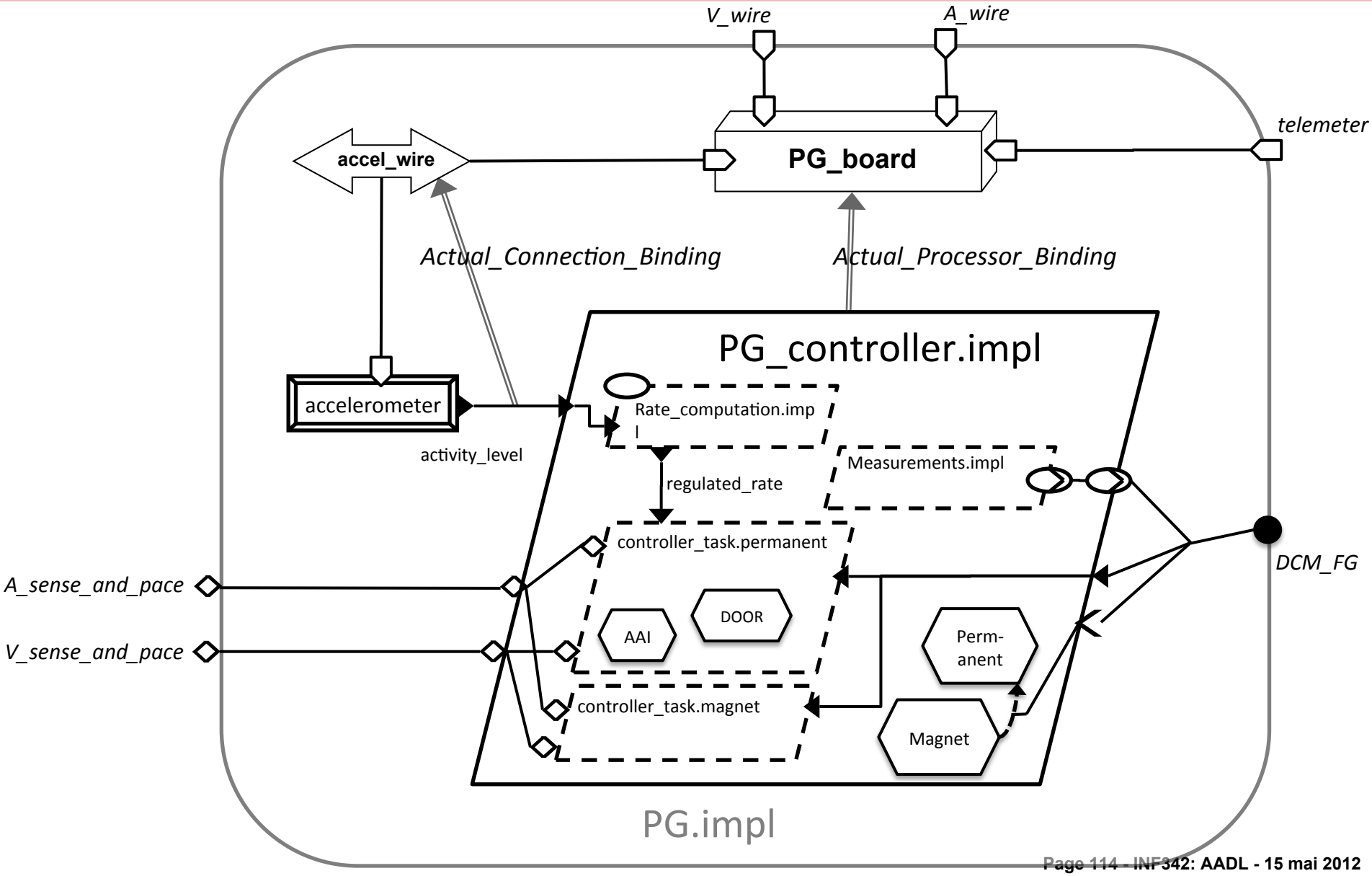
- PG: processus de controle



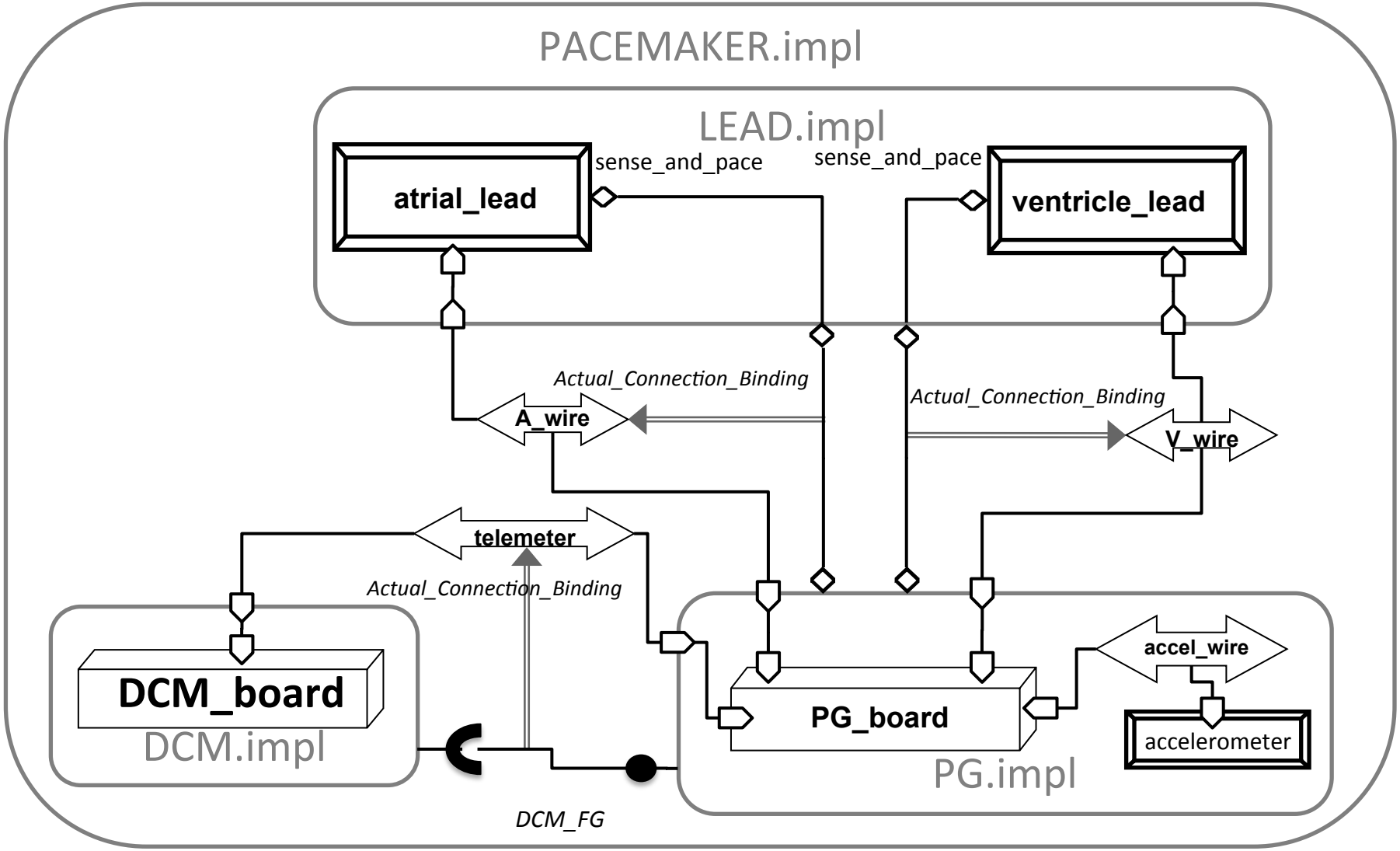
Défauts de la solution

- La solution proposée permet-elle d'assurer le respect des exigences temporelles?
- Quelque 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 information contenus 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

Structuration d'une modélisation AADL

Instanciation de l'architecture

- Une description AADL est une suite de déclarations
  - ⌘ analogie avec des diagrammes UML
  - ⌘ Exploitation (analyse et manipulation) limitées
    - ○ certaines propriétés sont associées à un sous-composant particulier, etc…
- 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
  - ⌘ les entités qui ne sont pas instanciées:
    - ○ les composants data associés aux features car il représentent un type.
  - ⌘ on manipule alors l'architecture telle qu'elle doit être dans la réalité

Organisation d'une description

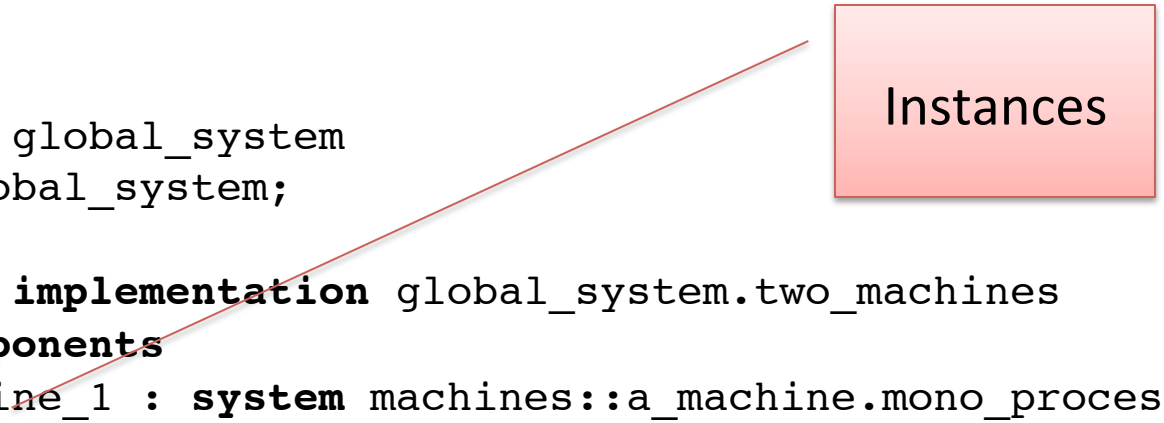
- Les paquetages (packages)
  - matérialisent des espaces de nom
  - une partie publique : visible de partout
  - une partie privée : uniquement visible depuis le paquetage
    - composants
    - groupes de ports
    - Propriétés
- Espace de nom anonyme (anonymous namespace)
  - le plus haut niveau de la description
    - paquetages
    - composants
    - groupes de ports
    - ensembles de propriétés
- 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;
```

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;
```

Composition de composants AADL

- Un composant peut avoir des sous-composants
  - ⌘ décrits dans les implémentations des composants
- Une modélisation AADL est une arborescence de composants

Catégorie	Peut contenir
data	data
thread	data
thread group	data, thread, thread group
process	thread, thread group
processor	memory
memory	memory
system	tous sauf subprogram, thread et thread group

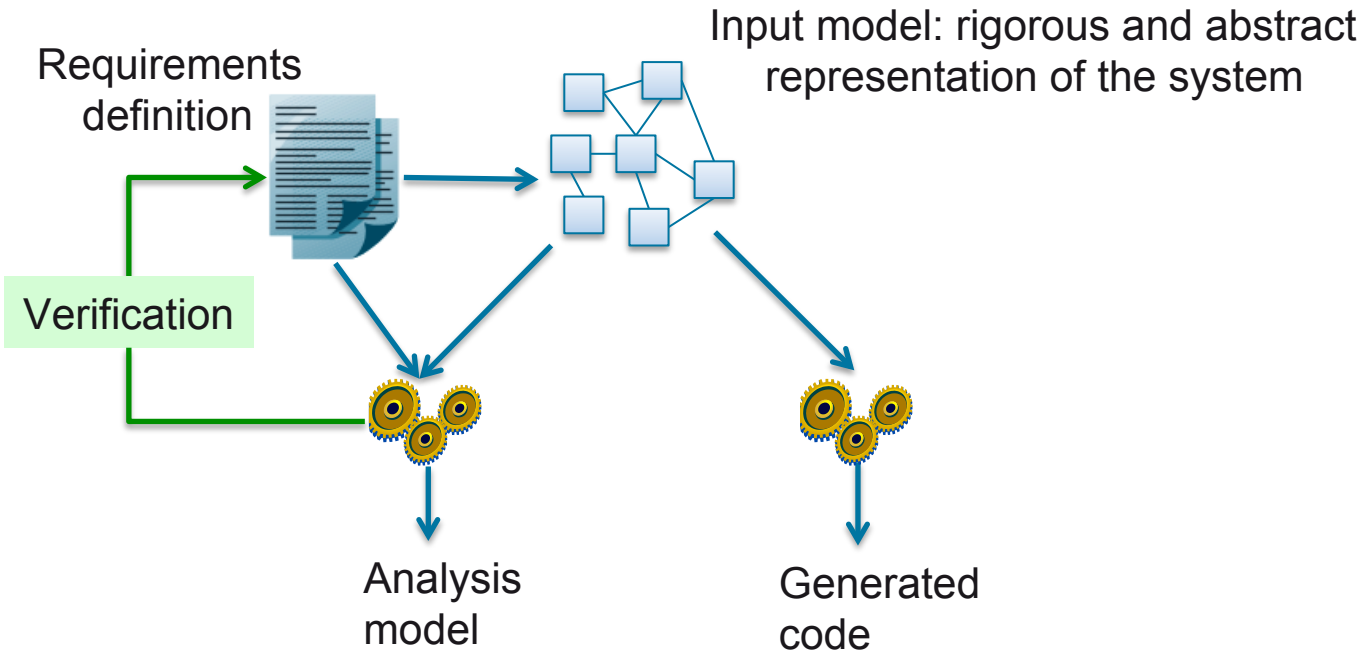
Exemple de composants

```
thread execution_thread
end execution_thread;

process a_process
end a_process;

process implementation a_process.one_thread
subcomponents
  thread1 : thread execution_thread;
end a_process.one_thread;

process implementation a_process.two_threads
subcomponents
  thread1 : thread execution_thread;
  thread2 : thread execution_thread;
end a_process.two_threads;
```

Génération de code, et analyse d'ordonnancement

MDE in a development process

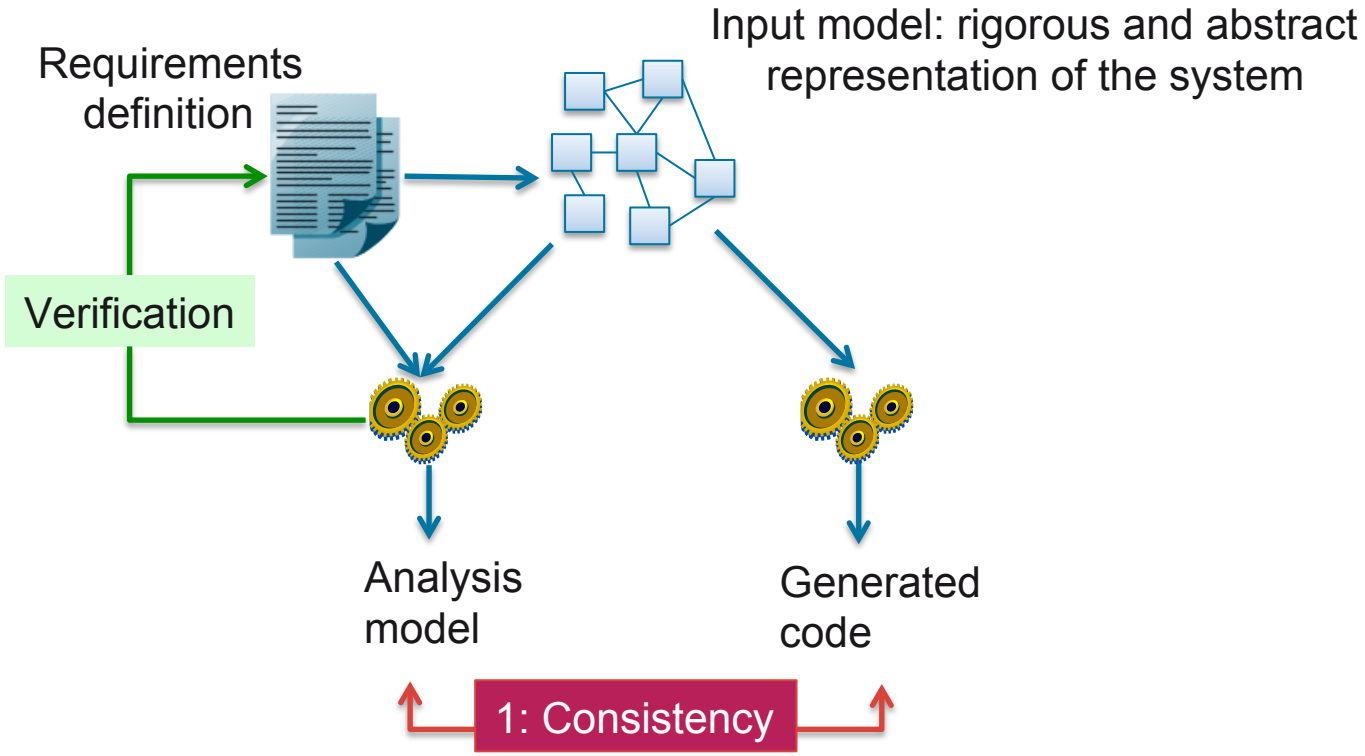
- Objectives of MDE: automate the development process



- Objectives of MDE in SCES:
  - ⌘ Improve confidence in the produced system
  - ⌘ Reuse components, and automate integration

Lessons learnt and research activities

- MDE and code generation for SCES requires to
  1. Check consistency between analysis results and produced system
  2. Produce alternative models and evaluate quality attributes



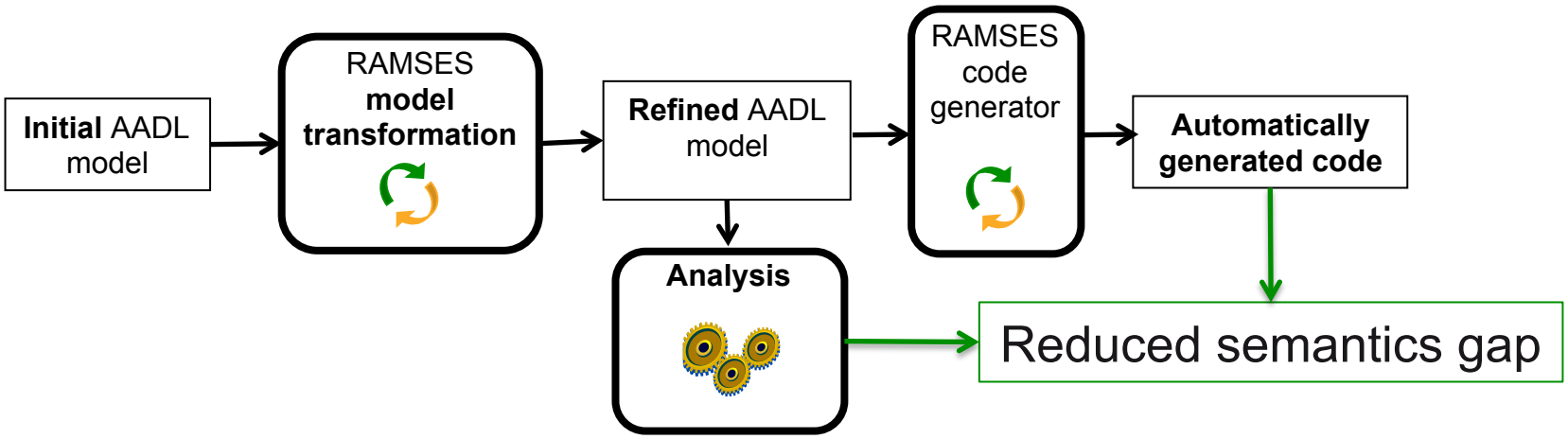
What is the effect of code generation?

- Code generation interprets an abstract model to produce its implementation:
  - Connections are mapped into queues, potentially using protected shared data, additional threads,
  - Operational modes may require additional threads to manage mode transitions,
  - Health monitoring requires faults detection and recovery mechanisms,
  - etc, etc.

➔ **Impacts quality attributes**

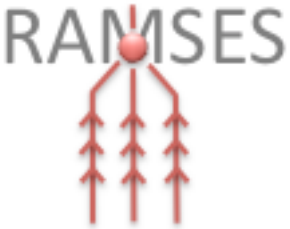
Method: models refinements

- **RAMSES: Refinement of AADL Models for Synthesis of Embedded Systems**
  - Collaborations with CMU/SEI and AMRDEC
  - Validation on an industrial case-study (THALES)



- PAPER(S) on model refinements for code generation
  - Design Patterns for Rule-based Refinement of Safety Critical Embedded Systems Models. *International Conference on Engineering of Complex Computer Systems (ICECCS'12)*
  - Architecture Models Refinement for Fine Grain Timing Analysis of Embedded Systems. *IEEE International Symposium on Rapid System Prototyping (RSP'14)*

What is RAMSES ?



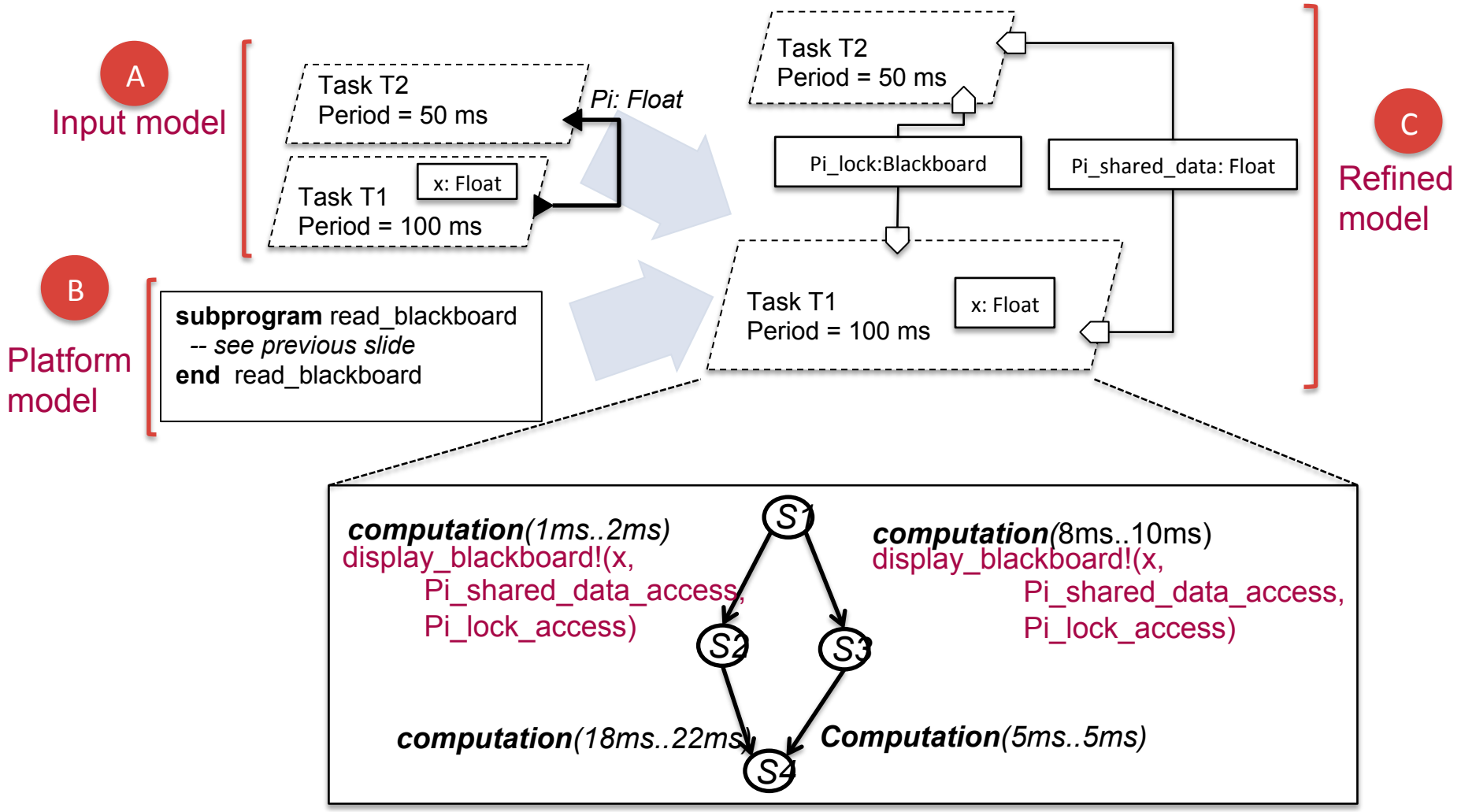
- An AADL model transformation framework integrated in OSATE
- A code generation plugin (C, Ada)
  - ARINC653 (avionics)
  - OSEK (automotive)
- A set of analysis plugins
  - Response time analysis (using AADL Inspector)
  - Memory consumption

Input and platform AADL models

- Input model
  - Interconnected periodic or sporadic threads with timing properties
  - State machines for threads behavior (with timing properties)
- Platform model
  - Data types and Subprograms of the underlying middleware or OS, with protection protocol for data accesses
  - State machines for subprograms behavior (with timing properties)

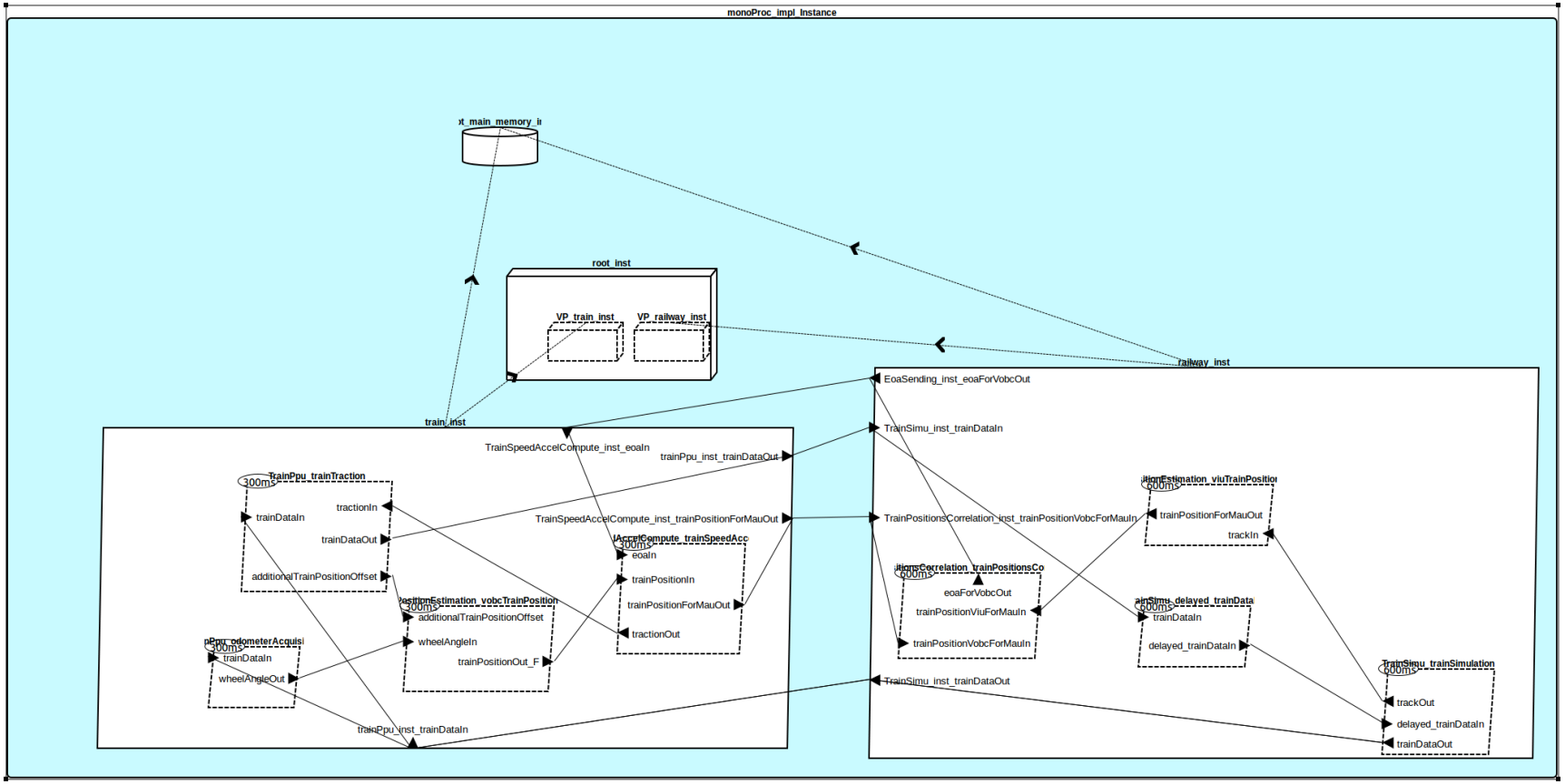
```
subprogram Read_Blackboard
features
  BLACKBOARD_ID: requires data access BLACKBOARD_ID_TYPE
                                {Concurrency_Control_Protocol => Priority_Ceiling; };
  …
annex behavior_specification {**
  states
    s: initial final state;
  transitions
    t: s-[]->s{computation(4 us .. 5 us) in bindings (RAMSES_Platform::POK_X86);
               BLACKBOARD_ID!<;
               computation(1ms..2ms) in bindings (RAMSES_Platform::POK_X86);
               BLACKBOARD_ID!>};
**};
end Read_Blackboard;
```

Architecture model refinement

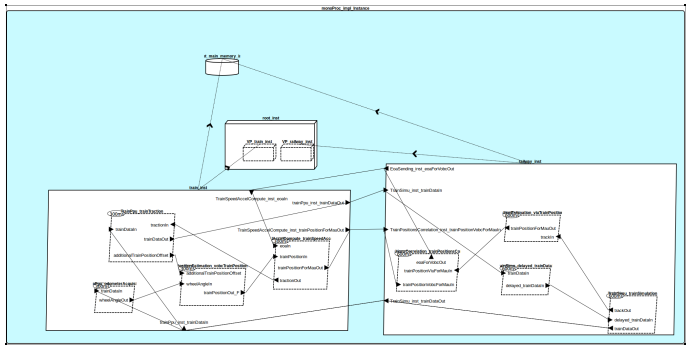


Use case: railway system

- **Objectives:**
  - Code generation
  - Worst-case response time analysis

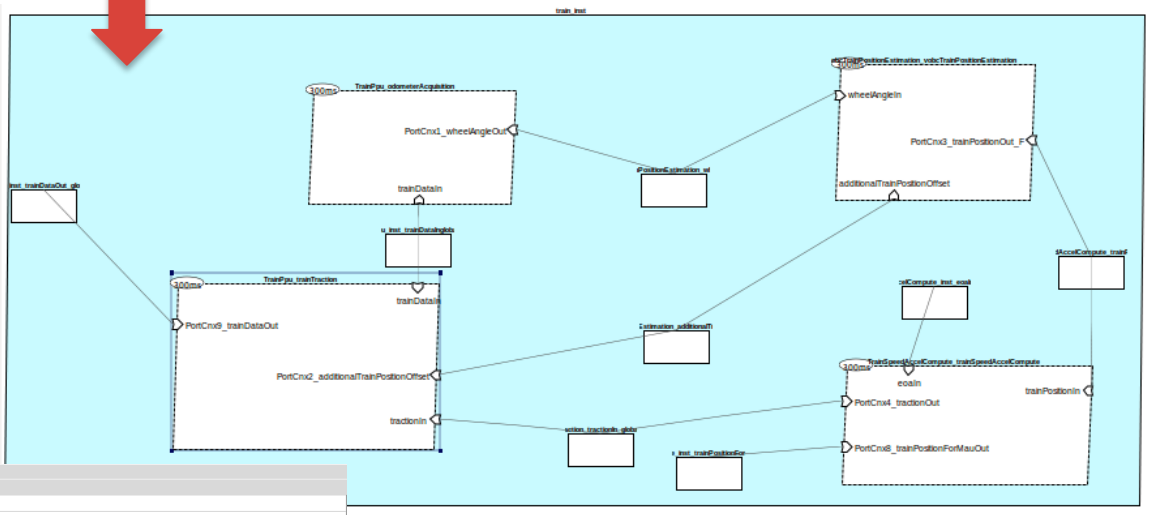


Experimentation results: fully automated process

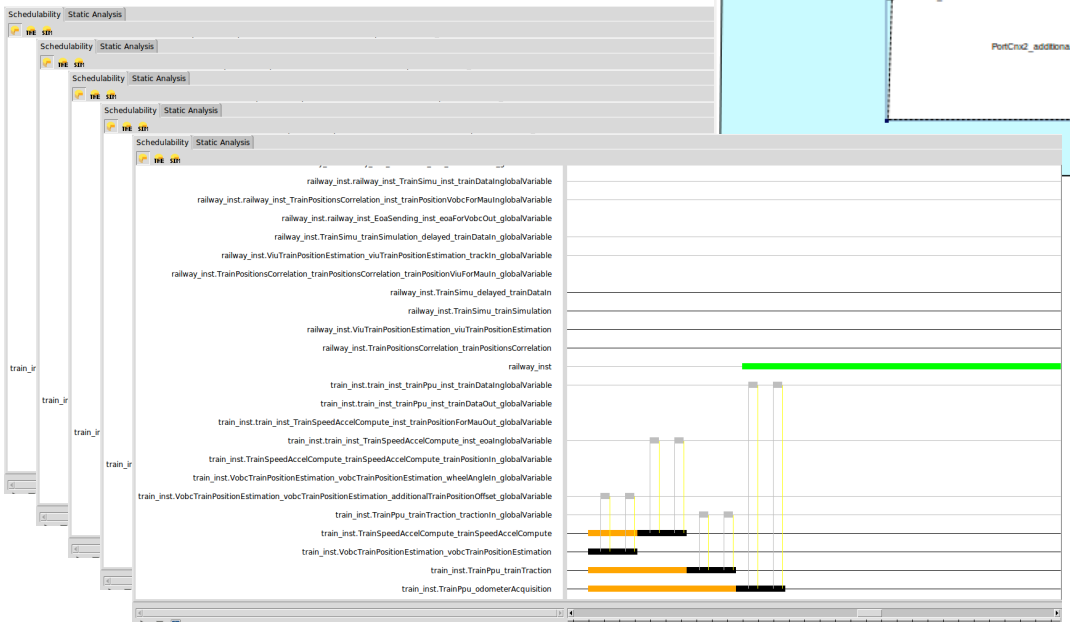


Design model

RAMSES refinement
AADL to AADL/ARINC653 model transformation



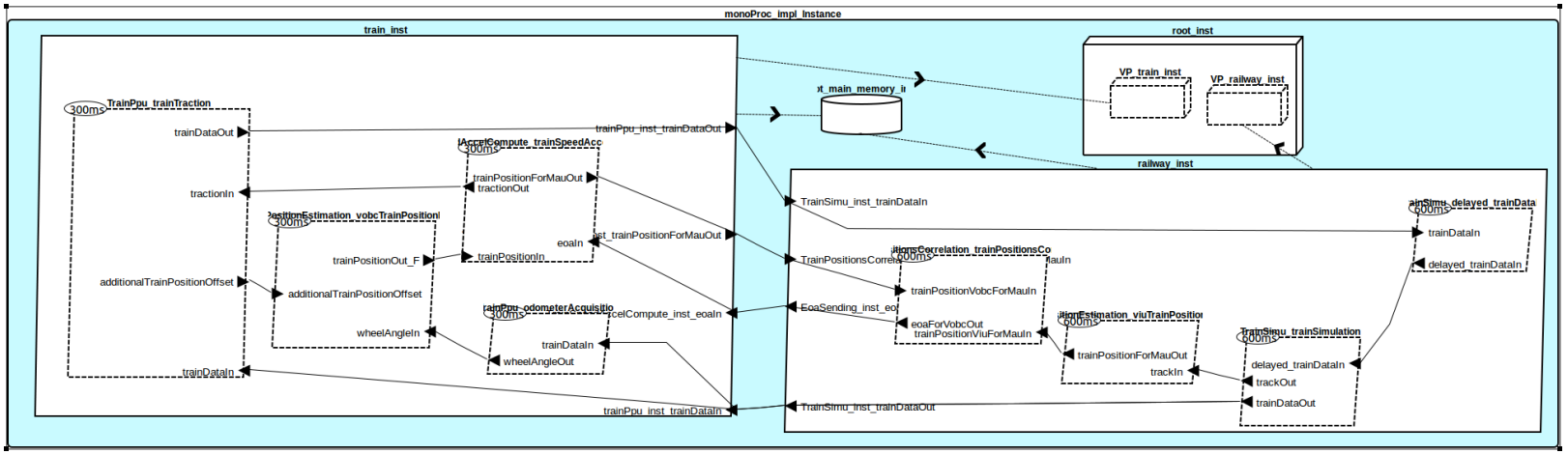
Implementation model



Analysis of all possible scheduling scenarii

Generated C/Ada code

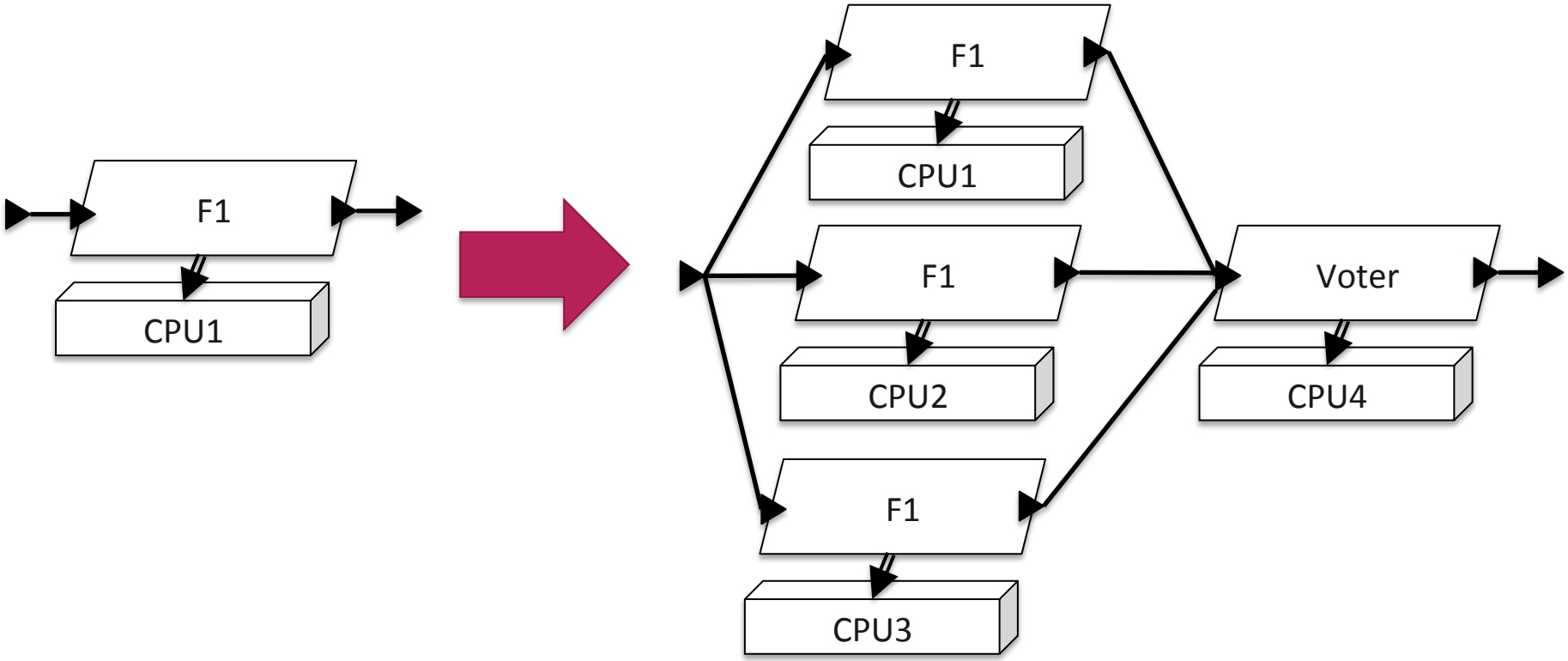
Case-study



- **This project was evaluated on a case-study of**
  - 2 partitions
  - 8 threads, two branches with locks per thread
- **Leading to 64 scheduling configurations to analyse**
  - Less than 3 minutes computation (parallelized, Intel Core i7-3740QM; 4 cores)

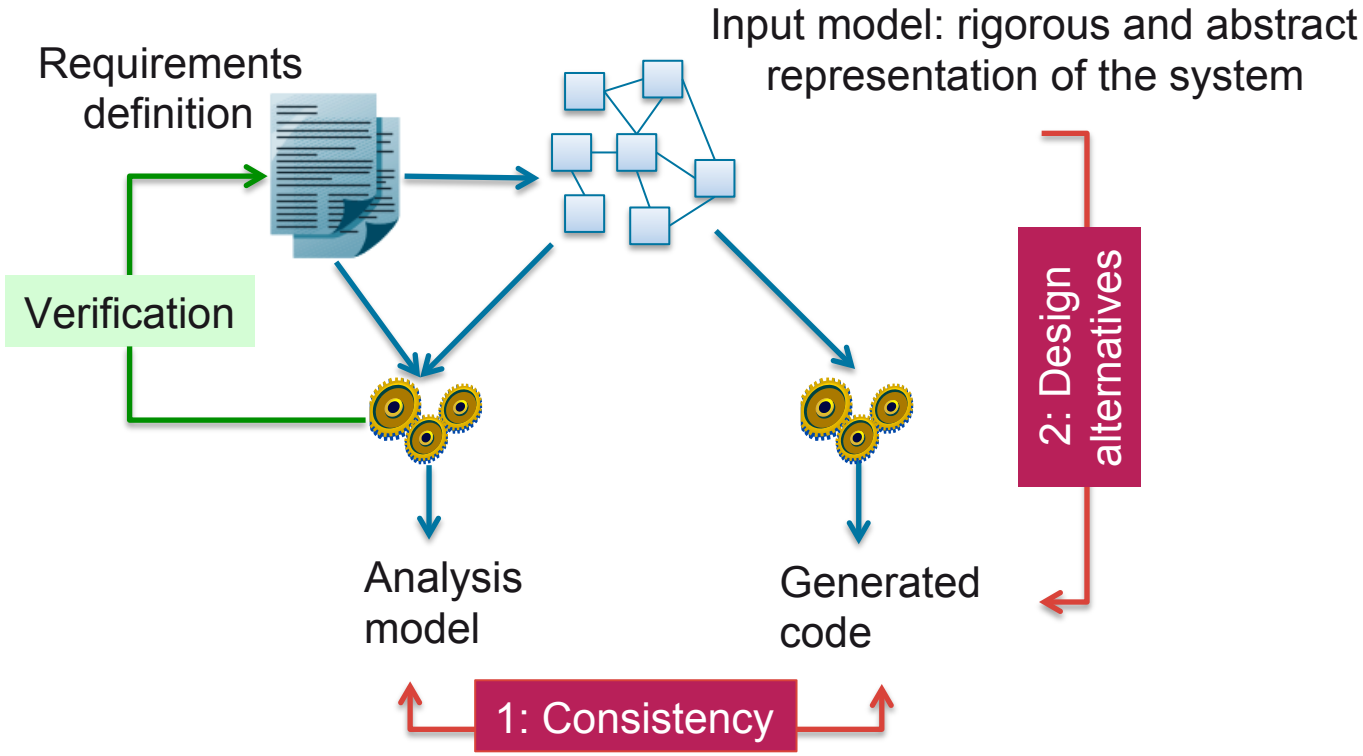
Ongoing work: safety and security design patterns as model transformations

- Example: tripple modular redundancy (simplified view)



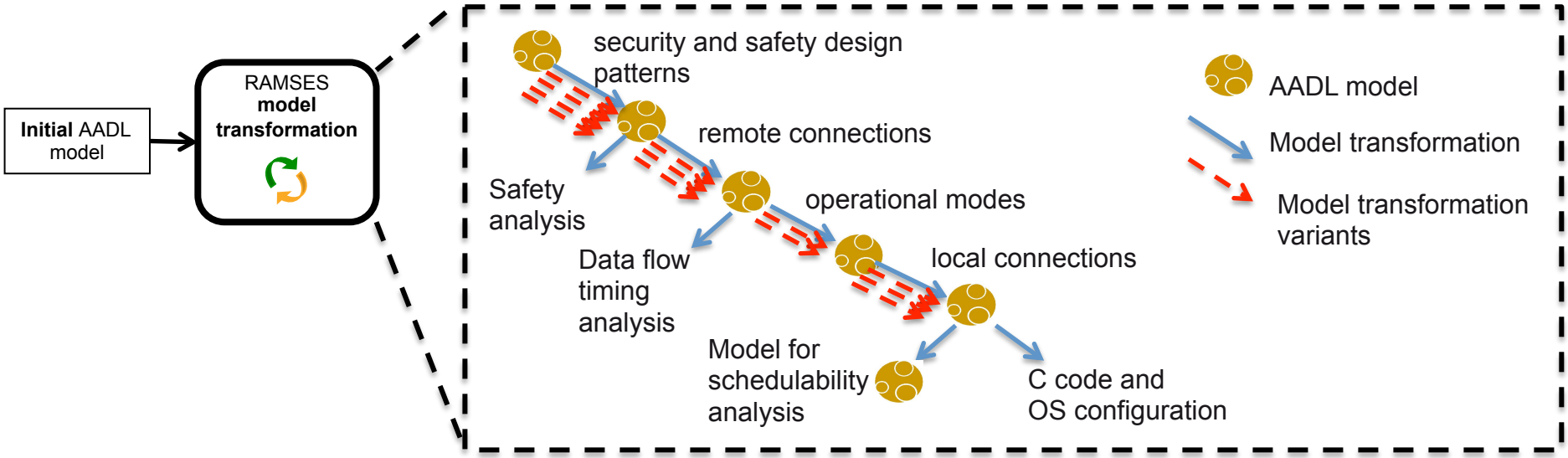
Lessons learnt and research activities

- MDE and code generation for SCES requires to
  1. Check consistency between analysis results and produced system
  2. Produce alternative models and evaluate non-functional properties



Model transformations and design space exploration

- Design alternatives exist when refining an abstract model into more concrete ones
  - Transformations must be broken down into transformation units to be chained
  - Variants of transformation units exist at each level of such chain



- **PAPER(S) on Selection and composition of model transformations**
  - An Automated Approach for Architectural Model Transformations. *22nd International Conference on Informations Systems Development (ISD'13).*
  - Automatic Production of Transformation Chains Using Structural Constraints on Output Models. *40th Euromicro Conference on Software Engineering and Advanced Applications (SEAA'14).*

Le rôle des annexes dans un modèle AADL

Les annexes

- Permettent d'enrichir la description en utilisant une syntaxe autre que celle d'AADL
  - ⌘ p.ex. Z, OCL, etc.
- Les annexes peuvent être contenues dans les composants et les groupes de ports
- Certaines annexes sont standardisées, d'autres peuvent être créées librement

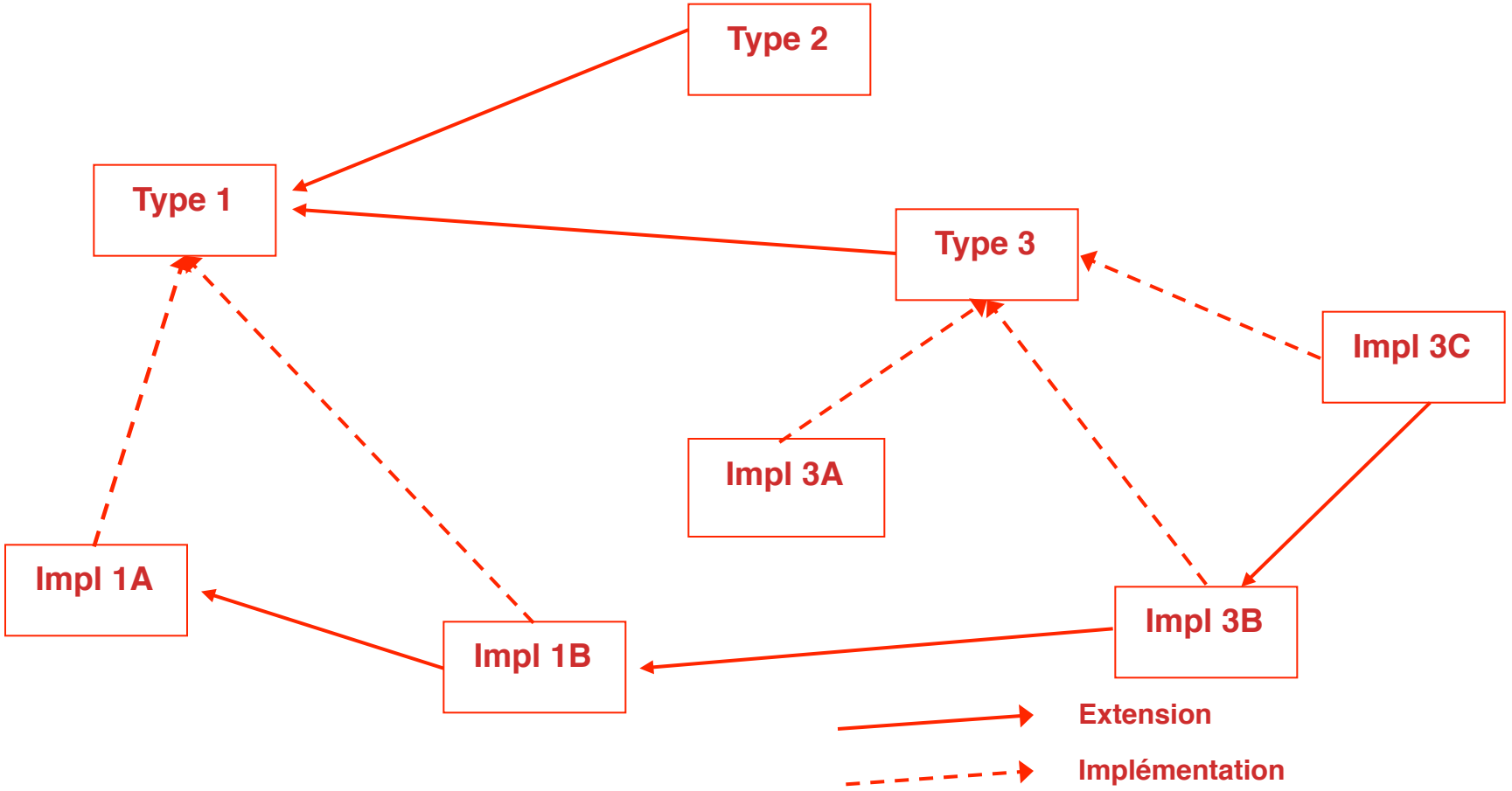
```
thread Collect_Samples
features
  Input_Sample    : in  data port Sample;
  Output_Average : out data port Sample;
annex OCL {**
  pre:  0 < Input_Sample < maxValue;
  post: 0 < Output_Sample < maxValue;
**};
end Collect_Samples;
```

Annexes et propriétés

- Deux façons d'enrichir une description AADL
  - utiliser des annexes
  - utiliser des propriétés
- Une annexe n'est pas obligatoirement interprétée par les outils d'exploitation
  - mais sa détection par un outils qui ne peut pas l'interpréter NE DOIT PAS causer des erreurs de syntaxe.
- On peut associer des propriétés à quasiment tous les éléments d'une description
- Une annexe sert à ajouter des précisions facultatives. Une propriété est généralement très liée à la description architecturale
- Les annexes et propriétés permettent une grande souplesse
  - attention à ne pas détourner leur utilisation
    - décrire tout le comportement d'un sous-programme avec des annexes au lieu d'utiliser les séquences d'appel
    - utiliser une propriété pour référencer un fichier qui contient la description en langage naturel de l'architecture d'un composant

Extension des composants

- un composant AADL peut étendre un autre (*component extension*)



Exemple d'extension de composant

```
thread execution_thread
end execution_thread;

process a_process
end a_process;

process implementation a_process.one_thread
subcomponents
  thread1 : thread execution_thread;
end a_processus.one_thread;

process implementation a_process.two_threads
extends a_process.one_thread
subcomponents
  thread2 : thread execution_thread;
end a_process.two_threads;
```

➔ a_process.two_threads contient **deux** threads

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;

PROCESSOR IMPLEMENTATION root.impl
SUBCOMPONENTS
  VP_train_inst : VIRTUAL PROCESSOR VP_train.impl;
  VP_railway_inst : VIRTUAL PROCESSOR VP_railway.impl;
PROPERTIES
  ARINC653::Partition_Slots => (300ms,300ms);
  ARINC653::Slots_Allocation => (REFERENCE(VP_railway_inst),
                                 REFERENCE(VP_train_inst));
END root.impl;
```

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? Periode, 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

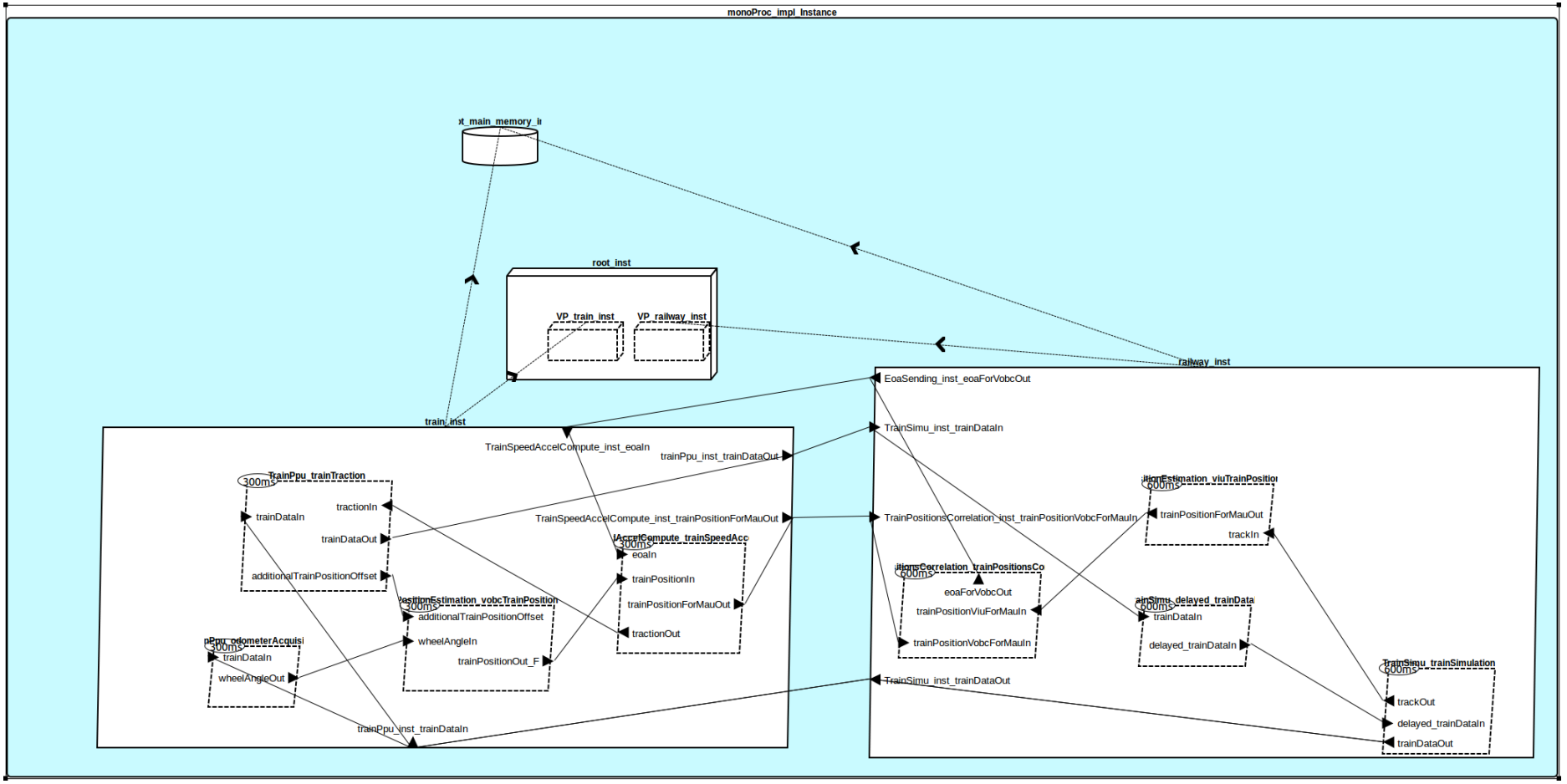
- Analyse à ce niveau de modélisation
  - ⌘ Ordonnançabilité gros grain (modèle de tâches, propriétés)
  - ⌘ Cohérence du dimensionnement (capacité/demande mémoire)
  - ⌘ Model-checking?
  - ⌘ Génération de code?

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? Periode, priorité, etc…)

- Que manque-t-il pour aller plus loin?
  - ⌘ Les canaux de communication,
  - ⌘ Le déploiement (mémoire et processeur)
  - ⌘ Comportement des tâches...

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

Exploitation de modèles AADL

AADL pour la génération d'un système exécutable

- Une modélisation AADL décrit
  - les caractéristiques des composants
  - les connexions
- Les propriétés standard permettent d'associer un code source à chaque composant
  - VHDL, …
  - Ada, C, …
- Ces codes sources indiqués en propriétés doivent se conformer aux spécifications AADL (signatures des sous-programmes…)
  - écrits à la main
  - générés automatiquement par d'autres outils
- On peut indiquer les codes sources des composants et générer une application pour un exécutif AADL

Paramètres pour la génération

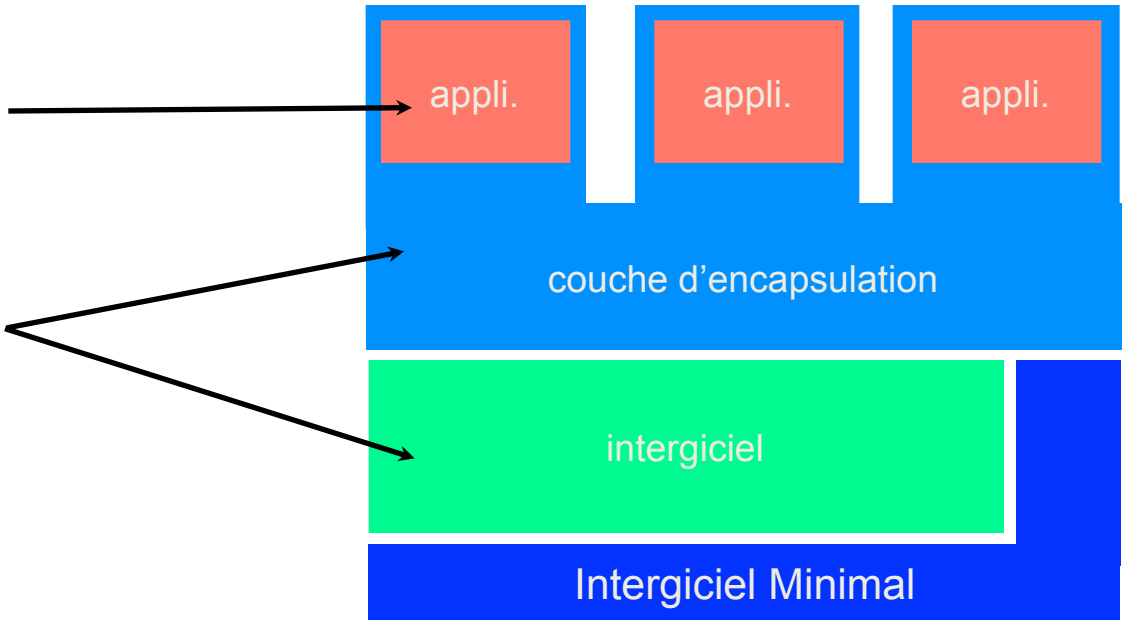
- Composants
  - matériels : fournissent les informations de déploiement
    - caractéristiques des machines
    - connexions sur les réseaux
    - …
  - logiciels
    - correspondent aux applications dont il faut générer le code
    - processus : modélisent les nœuds/partitions
    - threads : éléments actifs
    - sous-programmes : éléments réactifs des applications
  - systèmes
    - éléments de structure (non fonctionnels) de l'architecture
- Interfaces et connexions entre les processus
  - correspondent aux modèles de répartition utilisés

Exécutif pour les nœuds applicatifs

- L'exécutif AADL doit assurer deux tâches
  - ⌘ contrôle de l'exécution des threads (ordonnancement)
  - ⌘ prise en charge des communications (inter- et intra- nœuds)

Descriptions fonctionnelles (fournies par l'utilisateur)

Composants générés automatiquement à partir de la description AADL



Vérification de la description AADL

- Vérification syntaxique
  - valeur des propriétés bornées
  - ensemble de propriétés AADL_Project caractéristique des dimensions de l'exécutif
    - protocoles possibles, etc.
- Vérification de la cohérence des spécifications
  - tailles mémoire des données et des mémoires
  - temps d'exécution des sous-programmes et des threads
  - nombre de threads par processeur
- Vérification des limitations relatives à l'exécutif
  - bonne utilisation des propriétés vis-à-vis des capacités (resources) de l'exécutif
  - bonne utilisation des modes

Vérification de l'ordonnançabilité

- Au niveau des processus et des processeurs
  - ⌘ contrainte sur le nombre maximum de threads
  - ⌘ calcul de l'ordonnancement
- Exploitation des propriétés AADL
  - ⌘ protocole d'ordonnancement
  - ⌘ temps d'exécution
  - ⌘ communications entre les nœuds
- p.ex.: Cheddar

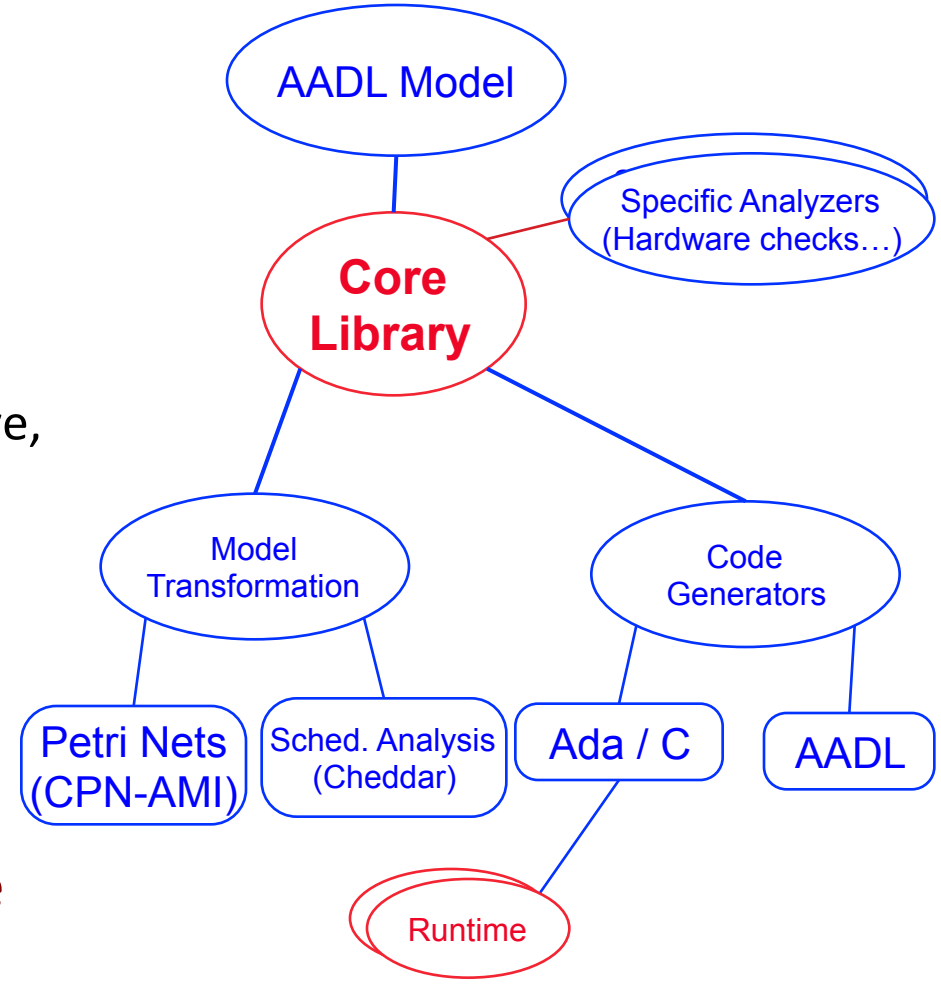
Vérification des flux d'exécution

- Vérification statique des connexions
- Utilisation de méthodes formelles

- p.ex.: réseaux de Petri
  - ⌘ pour étudier l'absence d'interblocages provoqués par l'assemblage des composants
  - ⌘ pour vérifier le dimensionnement des resources
  - ⌘ détecter les sous-ensembles architecturaux jamais utilisés
  - ⌘ …

Ocarina: un ensemble d'outils AADL

- Bibliothèque & outils pour manipuler AADL
  - ⌘ Parseurs & afficheurs AADL
  - ⌘ Vérification sémantique
- Générateurs de code
  - ⌘ Ada/PolyORB
  - ⌘ (Ada, C)/PolyORB-HI
  - ⌘ Pour plusieurs plateformes (Native, LEON, ERC32)
- Configuration du support d'exécution
- Vérification & Validation
  - ⌘ Réseaux de Petri
  - ⌘ Ordonnancement (Cheddar)
- Un outil en ligne de commande pour automatiser ces tâches



PolyORB-HI: Intergiciel pour les systèmes critiques

- Contraintes temps réel dur spécifiques aux systèmes critiques:
  - Modèle de concurrence analysable : Profil Ravenscar
  - Restrictions du langage de programmation pour les systèmes critiques
    - Encore plus restrictif que le profil Ravenscar, ex. pas d'allocation dynamique ni d'orienté objet
- PolyORB-HI: un support d'exécution AADL
  - Supporte les constructions AADL
    - Threads périodiques et sporadiques, données, etc.
  - Configuré automatiquement à partir du modèle AADL
    - Ressources calculées et allouées statiquement
    - Pas d'intervention requise de la part de l'utilisateur
  - Occupe une faible taille en mémoire
    - Toute la valeur ajoutée est dans la phase de génération de code: Contribue à la thématique des "usines à intergiciels"

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:
    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
  - http://beru.univ-brest.fr/~singhoff/cheddar/index-fr.html