



# AADL: Architecture Analysis & Design Language

Etienne Borde, Gilles Lasnier, Bechir Zalila,

Laurent Pautet, Thomas Vergnaud

[{nom.prenom}@telecom-paristech.fr](mailto:{nom.prenom}@telecom-paristech.fr)

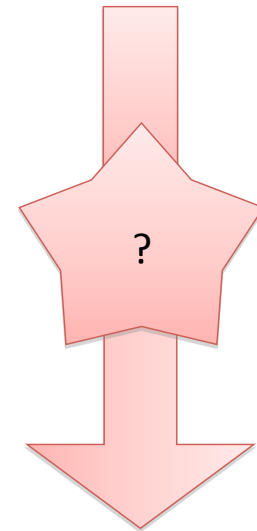


## Introduction et généralités

# Contexte (1)

- **Modèle formel**
  - ⌘ SCADÉ, SDL, réseaux de Petri, etc.
  - ⌘ couplés à des générateurs de code
  - ⌘ mise en place de la répartition
- **Éléments d'implantation**
  - ⌘ réseaux, intergiciels, OS, etc.
  - ⌘ contraintes sur l'application répartie
- **Comment rassembler ces deux aspects?**
  - ⌘ passer de la description formelle à l'implantation
  - ⌘ évaluer les performances de l'application
    - temps d'exécution
    - empreinte mémoire
    - fiabilité

modélisation de haut  
niveau

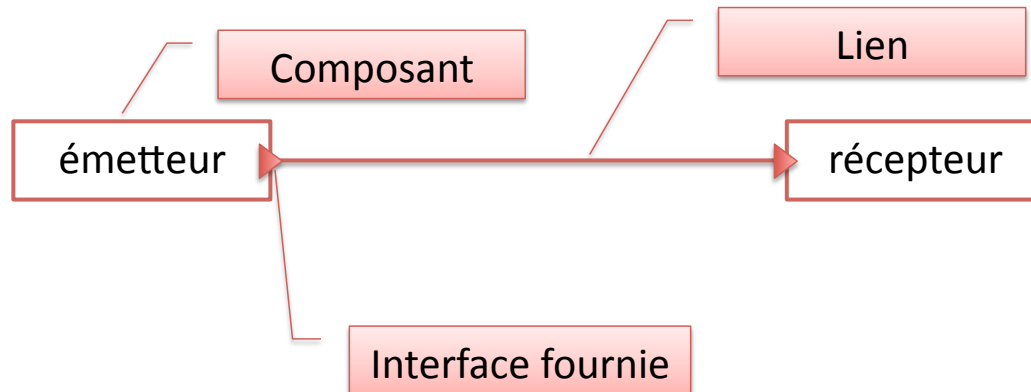


implantation

- **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

- 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  
à récepteur

- **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

- 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	Difficile à extraire l'information
Graphical	Facile à lire, à partager	Dépendance aux outils

- 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





## Les principes de base d'AADL

# Pourquoi AADL?

- ADL spécifique aux systèmes temps-réels:
  - ⌘ Certaines propriétés pré-définies décrivent le comportement du système (politique d'ordonancement, etc...)
  - ⌘ Certaines propriétés permettent de représenter les allocations 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'ordonnabilité, vivacité, etc...)

- 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 niveaux 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
  - ⌘ Dédié aux systèmes répartis embarqués temps-réel
  - ⌘ Décrit des éléments matériels et logiciels
  - ⌘ Conçu pour permettre la génération de systèmes exécutables
    - expression des caractéristiques des éléments du système
    - traduction en langage de programmation
- Plusieurs représentations
  - ⌘ représentation textuelle
    - pour contrôler tous les détails du système
  - ⌘ représentation en XML
    - pour l'interopérabilité entre outils
  - ⌘ représentation graphique
    - convenable pour avoir une vision globale du système
  - ⌘ profil UML 2, et représentation en UML 1.4
- Version 1.0 publiée en 2004
- Version 2 a été publiée fin 2009

# AADL 1.0 (<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
  - ⌘ Ocarina : suite d'outils pour générer des applications (ENST)
    - noyau central pouvant être intégré dans différentes applications
    - 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

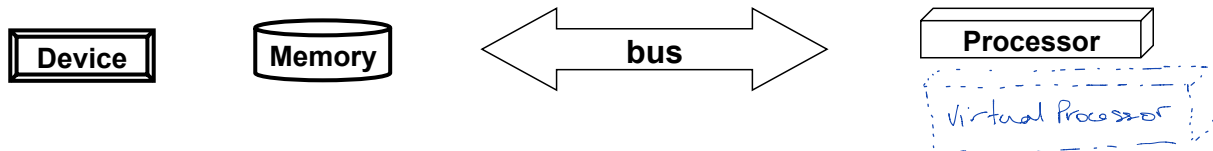
- Composants concrets
  - ⌘ Catégorie (Thread, data, bus, processor, etc...)
  - ⌘ Type : définition des interfaces (ports)
  - ⌘ Implémentation : définition de la structure interne des composants (sous-composant, code, spec. comportementale, etc...)
  - ⌘ Connections : relient les interfaces de sous-composants
- 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

- 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
  - ⌘ 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



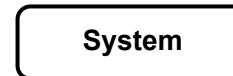
- Plusieurs catégories

- ⌘ 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.



- Permet de structurer la description (matériel+logiciel)
- Contient les composants qui peuvent être manipulés de façon indépendante :
  - ⌘ system
  - ⌘ processor, memory, device, bus
  - ⌘ process, data
- **MAIS PAS :**
  - ⌘ thread
  - ⌘ thread group
  - ⌘ subprogram



# 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

<u>Catégorie</u>	<u>Peut contenir</u>
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;
```

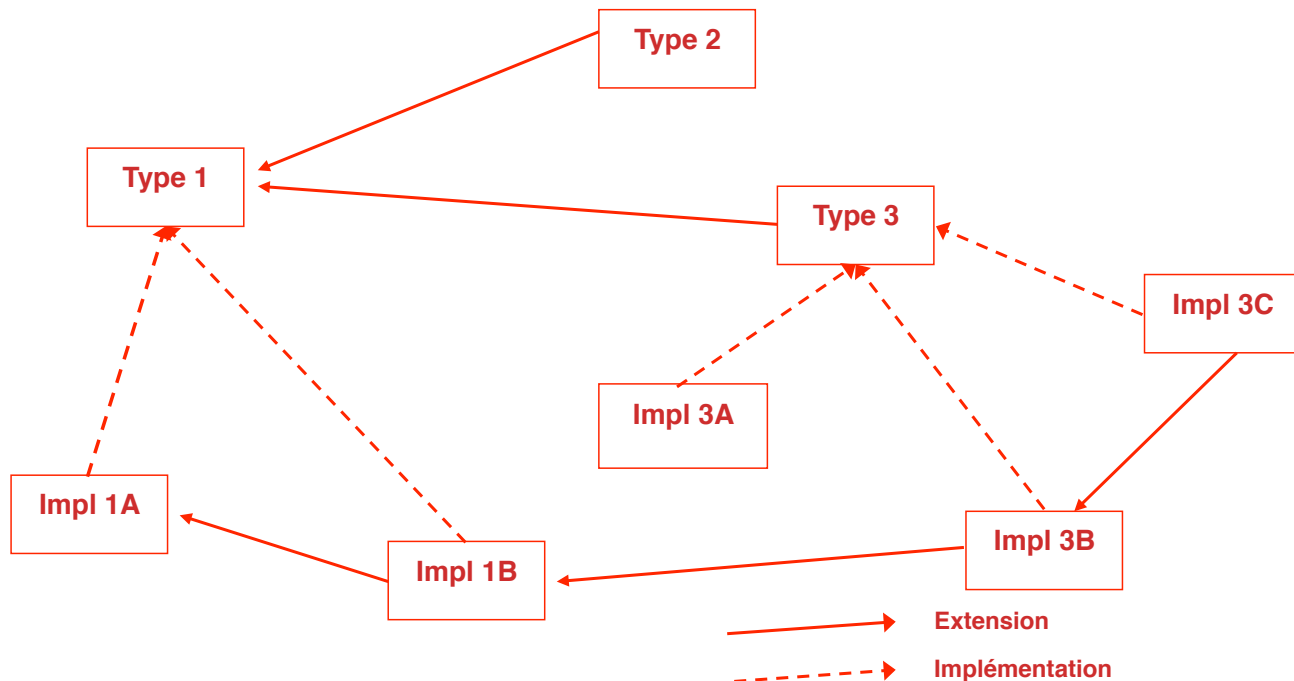
```
process a_process  
end a_process;
```

```
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;
```

# 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



## 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 d'un signal
  - ⌘ event data : signal + données ; comparable à un message
- les ports peuvent être déclarés en
  - ⌘ entrée (in)
  - ⌘ sortie (out)
  - ⌘ entrée-sortie (in out)


- 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

Représentation 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

Représentation graphique 

# Features: les accès aux composants - 2



- On exprime ainsi l'obligation de brancher un composant avec un autre pour obtenir un système cohérent
  - ⌘ 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 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;
```

# 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;
```

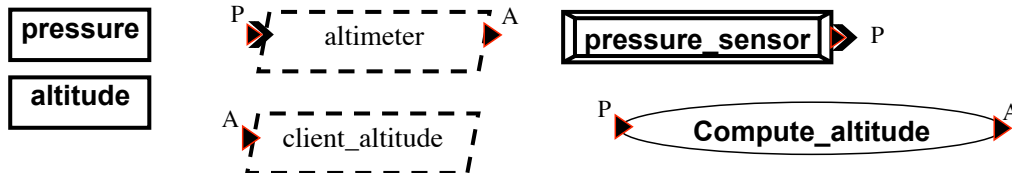
```
data altitude  
end altitude;
```

```
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;
```





## Les connexions des composants AADL

- Pour relier les « features » entre elles
  - ⌘ ports
  - ⌘ paramètres
  - ⌘ sous-programmes d'interface
  - ⌘ accès aux sous-composants
  - ⌘ groupes de ports
- Les connexions ont une direction
  - ⌘ les entrées sont reliées aux sorties de sous-composants
  - ⌘ les entrées d'interfaces sont reliées aux entrées des sous-composants ;  
inversement pour les sorties.
- 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 »

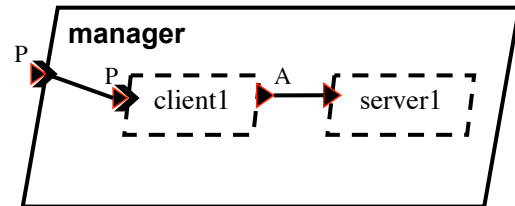


# Exemple de connexion (1)

```
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;    -- transmission 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;
```

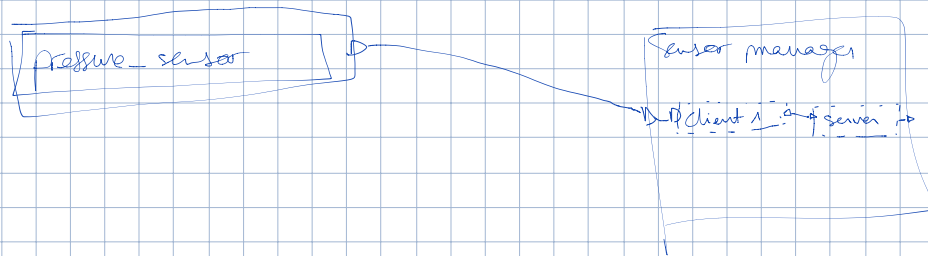
# Exemple de connexion (3)

```
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;
```





## Le rôle des propriétés et annexes dans un modèle AADL

- 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_processor.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;
```

- 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.

- 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, PTP, PCP, ICPP, etc...).

- 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>

# 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;
  
```



- 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



## Structuration d'une modélisation AADL

- 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;
```

# Exemple 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

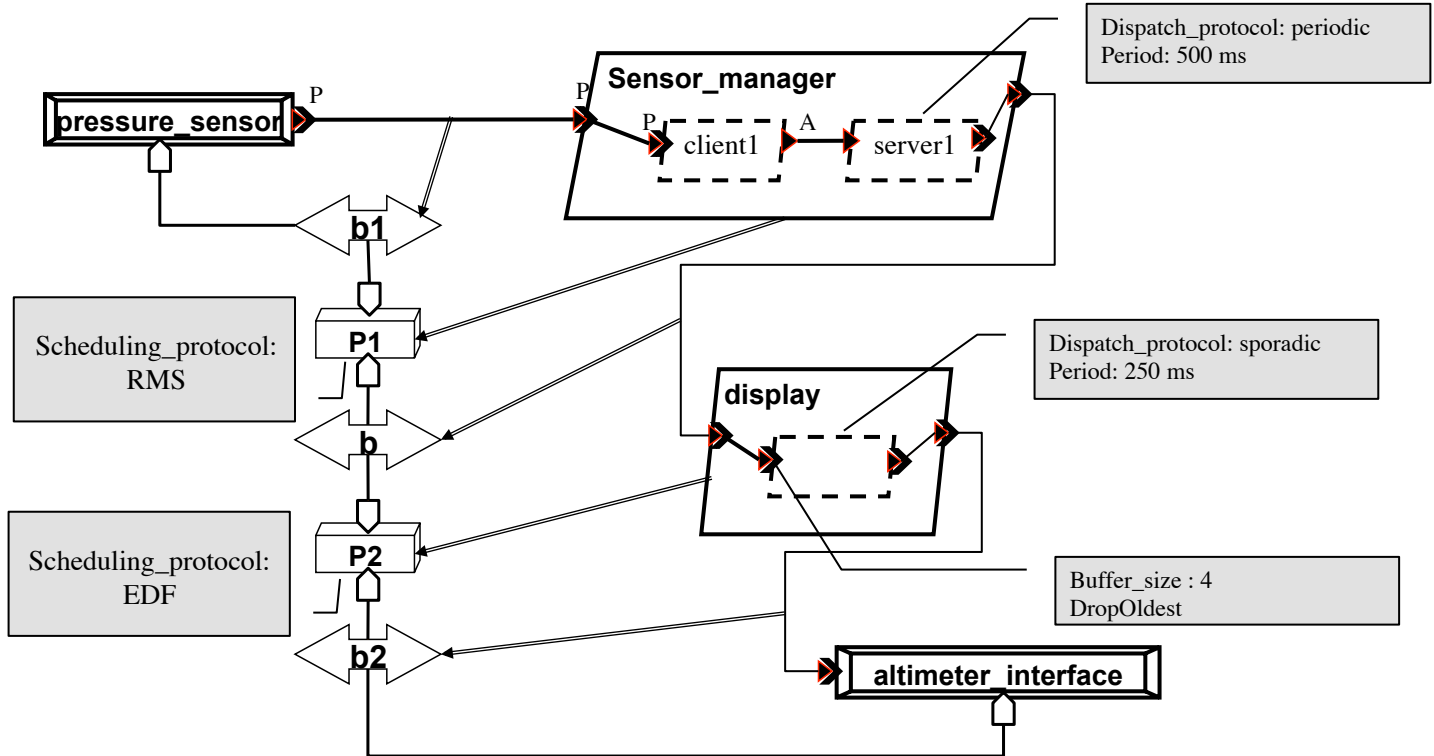
```
system global_system  
end global_system;
```

```
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

# Exemple complet: altimètre





## Modélisation du flux d'exécution 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;
```

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

# 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 une branche d'un if/then/else 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 *Deque Protocol*)
- `p'count` et `p'fresh` pour connaître 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!  
};

\*\*};

**end** controller\_task.impl;

# 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 *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
```

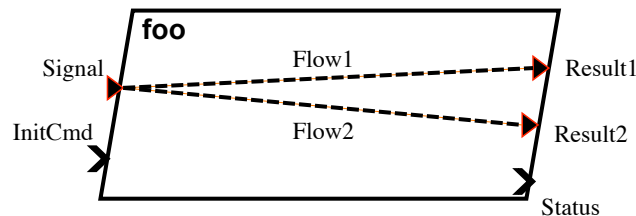
```
  Flow1 : flow path   Signal -> Result1;
```

```
  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;
```



## Altération d'une architecture AADL à l'exécution

- 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 propriétés peut dépendre du mode courant
- Les changements de modes permettent de représenter des architectures dynamiques
  - ⌘ évolution dans la configuration de l'architecture
  - ⌘ ensemble déterminé 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;
```



## 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
- Une annexe du standard donne les correspondances entre les syntaxes AADL et Ada ou C
- On peut indiquer les codes sources des composants et générer une application pour un exécuteur 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

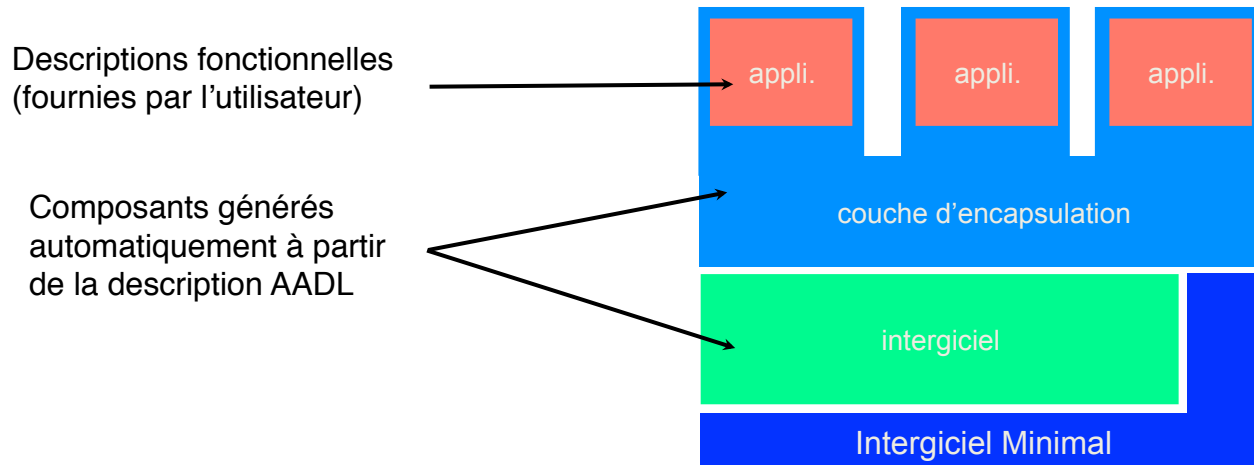
- **Interfaces et connexions entre les processus**

- ⌘ 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)



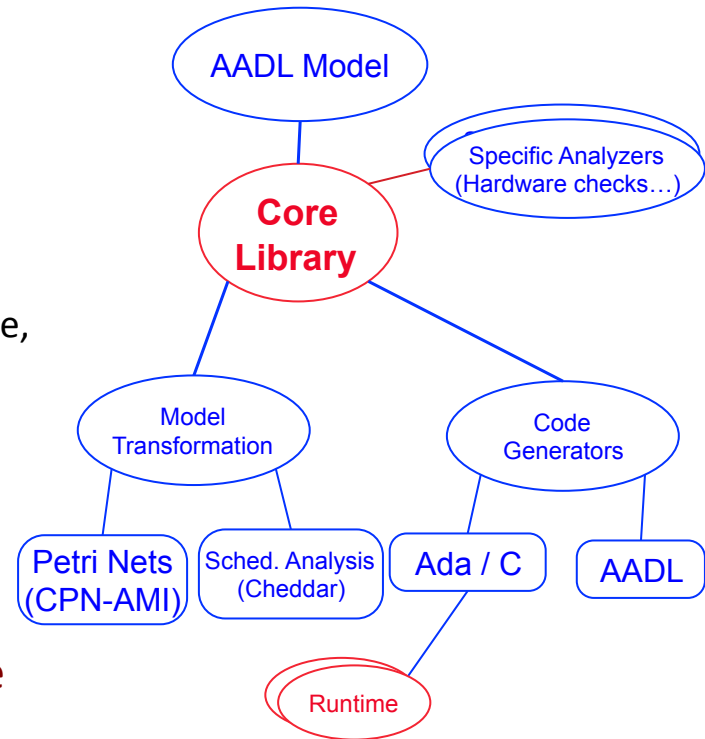
- **Vérification syntaxique**
  - ⌘ valeur des propriétés bornées
  - ⌘ ensemble de propriétés AADL\_Project caractéristique des dimensions de l'exécutif
    - protocoles possibles, etc.
- **Vérification de la cohérence des spécifications**
  - ⌘ tailles mémoire des données et des mémoires
  - ⌘ temps 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 de l'exécutif
  - ⌘ bonne utilisation des modes

- 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 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 la définition correcte des données
  - ⌘ 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



- **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
  - ⌘ 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.
  - ⌘ Configure 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
  - ⌘ Contribuer à la thématique des “usines à intergiciels”

- **Modélisation concrète**
  - ⌘ dernière phase avant la génération/déploiement du système
  - ⌘ précision dans la modélisation
  - ⌘ exploitation pour la vérification et la génération de prototypes
  - ⌘ possibilités de corrections sur l'architecture
- **AADL offre une grande souplesse de modélisation**
  - ⌘ degré de modélisation selon les besoins
  - ⌘ peut être utilisé comme langage fédérateur
    - exploitation par plusieurs outils différents
- <http://aadl.telecom-paristech.fr/ocarina>
- <http://aadl.telecom-paristech.fr/polyorb-hi>
- <http://beru.univ-brest.fr/~singhoff/cheddar/index-fr.html>
- <http://www-src.lip6.fr/logiciels/mars/CPNAMI/>