AADL

un langage pour la modélisation et la génération d'applications

Thomas Vergnaud, thomas.vergnaud@enst.fr

Les langages de description d'architecture

- la conception des systèmes devient complexe
  - difficulté de compréhension du système
  - il faut s'assurer de la cohérence de l'architecture du système
- un formalisme de description est nécessaire
  - pour décrire l'architecture
  - pour analyser et vérifier ses propriétés

- ADL formels
  - pour formaliser la description du fonctionnement d'un système
  - pas destinés à s'intégrer dans une démarche de génération automatique
  - Wright, Rapide
- ADL restreints
  - pour décrire l'assemblage de composants logiciels ou la cohérence de l'application
  - ArchJava, Fractal
- ADL concrets
  - pour décrire l'architecture afin de la générer automatiquement
  - UML, AADL

Architecture Analysis & Design Language

- normalisé par la SAE (Society of Automotive Engineers)
  - depuis 2004 : AADL 1.0
  - 2006 : AADL 2.0
- évolution de MetaH, qui était développé par Honeywell
  - pour les systèmes embarqués temps-réel
  - conçu pour permettre la génération de systèmes exécutables
    - *expression des caractéristiques des éléments du système*
    - *traduction en langage de programmation*
  - description des éléments matériels et logiciels
- plusieurs représentations
  - texte
  - XML
  - notation graphique
  - profil UML 2, et représentation en UML 1.4
- déjà utilisé par de grands projets
  - COTRE (Airbus)
  - ASSERT (ESA, EADS, ENST, INRIA, etc.)
  - …

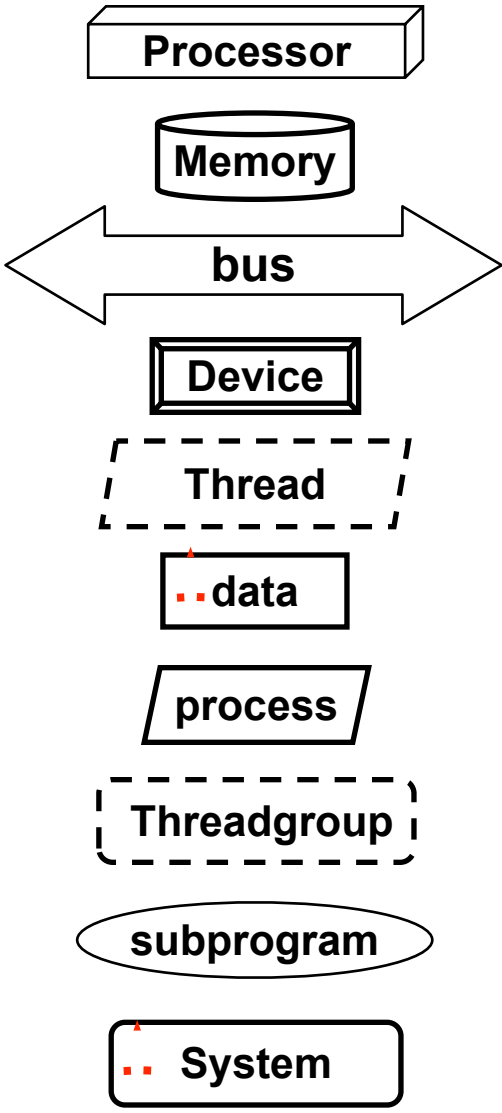
Principes généraux

- des **composants** reliés entre eux par des **connexions** qui relient leurs **interfaces**. Chaque composant est défini par un **type**, et une ou plusieurs **implémentations**.
- les composants peuvent contenir des **sous-composants**.
- il est possible d'attribuer des **propriétés** à chaque élément (composant, sous-composant, connexion, port, etc.)
- les caractéristiques (p.ex. sous-composants, propriétés, etc.) de l'implémentation peuvent varier selon des **modes** de configuration

- une description AADL est très concrète, pour permettre de générer facilement un système fonctionnel à partir de sa description. Tous les éléments d'une description ont une sémantique précise.
- AADL est un langage descriptif : les éléments peuvent être donnés dans n'importe quel ordre.

Les composants

- éléments de base d'une description architecturale
- déclarations en plusieurs parties
  - *component type* : interface
  - *component implementation* : structure interne
  - un type peut avoir plusieurs implémentations
- plusieurs catégories
  - matériels (*execution platform component*)
    - *processor, memory, bus, device*
  - logiciels (software components)
    - *thead, thread group, process, data, subprogram*
  - systèmes (*systems*)
    - *peuvent contenir d'autres composants*
    - *structurent l'architecture*
- les composants peuvent être déclarés dans des packages
  - structuration de la description
- les composants peuvent avoir des sous-composants
  - règles de composition



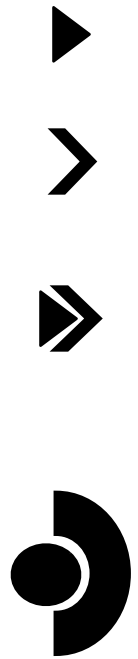
Exemple de composants


```
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_threadss;
```

```
subprogram prog1 end prog1;
subprogram prog2 end prog2;
thread a_thread end a_thread;

thread implementation a_thread.example
calls {
  call1 : subprogram prog1;
  call2 : subprogram prog2;
  call3 : subprogram prog1;
};
end a_thread.example;
```

Les interfaces (*features*)

- les ports permettent de modéliser les échanges d'information
  - 3 catégories de port :
    - *data : transportent des données ; comparables aux états d'un circuit électronique*
    - *event : semblables aux signaux*
    - *event data : des signaux qui transportent des données ; comparables aux messages*
  - peuvent être déclarés *in*, *out* ou *in out*
- les paramètres
  - pour les sous-programmes
  - peuvent être reliés à des *data ports* ou des *event data ports*
- sous-programmes d'interface
  - fournis par un thread : modélisation d'un RPC
  - fournis par une donnée : modélisation d'une classe
- accès aux sous-composants
  - requis ou fournis
  - concerne les bus et les données
- groupes de ports
  - pour agréger des interfaces



Les connexions

- pour relier les interfaces entre elles
  - ports
  - paramètres
  - sous-programmes d'interface
  - accès aux sous-composants
  - groupes de ports
- les connexions ont une direction
  - les entrées des sous-composants sont reliées aux sorties d'autres sous-composants
  - les entrées d'interfaces sont reliées aux entrées des sous-composants ; idem pour les sorties
- les features de sortie
  - peuvent être « 1 vers *n* »
- les features d'entrée
  - sont « 1 vers 1 »
  - event data ports & event ports entrants
    - *« n vers 1 »*
    - *car gestion de files d'attente*

Exemple de *features* & connexions


```
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;
```

```
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;
```

Les modes & les flots

- les modes permettent de modéliser la reconfiguration du système en fonction d'événements
  - définis au niveau des implémentations 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
  - ensemble déterminé de configurations possibles
- 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
  - les *component types* contiennent
    - des spécifications de flots (*source, sink* et *path*)
  - les *component implementations* contiennent
    - des implémentations de flots (*source, sink* et *path*)
    - des flots de bout en bout (*end to end*)

Les propriétés

- une propriété permet d'**associer** une **valeur** d'un certain **type** à un **nom**
- les propriétés sont définies dans des **ensembles de propriétés** (*property sets*)
  - ensemble de propriétés standard défini par la norme
  - il est possible de définir de nouveaux ensembles de propriétés
  - plusieurs types possibles
    - *booléen, entier, réel, chaîne de caractères, valeur parmi une énumération, catégorie d'élément (composant, connexion, etc.), plage de valeurs*
    - *définition d'unités*
  - une propriété peut ne s'appliquer qu'à certaines catégories d'éléments (ex. processeurs, connexions, etc.) ou à toutes
- une propriété peut être associée
  - au niveau de la déclaration d'un composant
  - au niveau de son instanciation comme sous-composant
  - à un sous-sous-composant

Exemples d'association de propriété


```
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_Swap_Execution_Time => 0 ms .. 5 ms;};
  process1 : process a_process;
properties
  Actual_Processor_Binding => processor1 applies to process1;
end a_process.simple;
```

Description complète

- une modélisation AADL est un ensemble de déclarations de composants
  - les sous-composants sont des instances des déclarations
  - analyse et manipulation limitées
    - *certaines propriétés sont associées à un sous-composant particulier*
    - *il est difficile d'appréhender l'architecture complète*
- pour travailler sur l'architecture, il est nécessaire d'**instancier** la modélisation
  - un système doit jouer le rôle de racine pour l'architecture
    - *pas d'interface*
    - *une implémentation qui contient les sous-composants de l'architecture*
  - les sous-programmes ne sont pas des sous-composants
    - *instanciés une seule fois dans leur package*
    - *les composants data associés aux ports ne sont pas instanciés non plus*
  - arborescence d'instances de composants correspondants aux déclarations de sous-composants

```
system global_system end global_system;
```

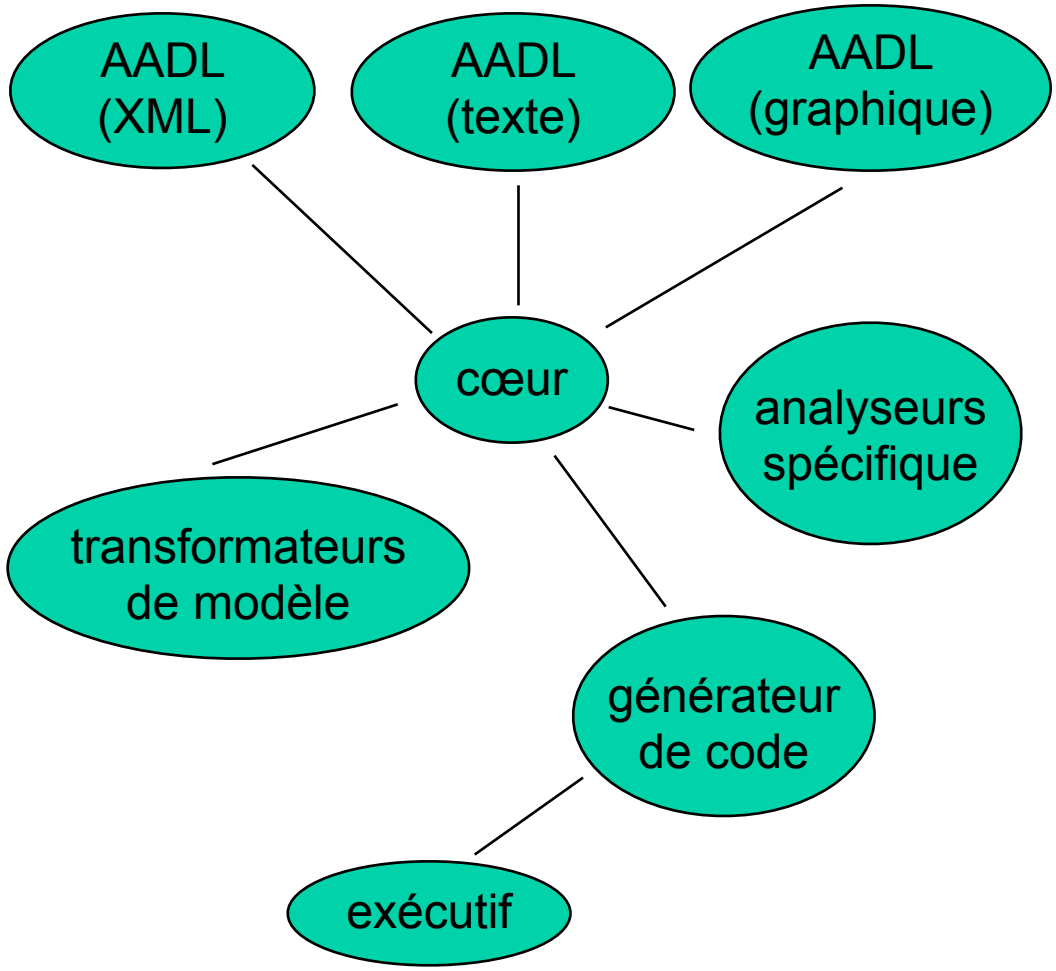
```
system implementation global_system.two_machines
subcomponents
  machine_1 : system a_machine.mono_processor;
  machine_2 : system a_machine;
end global_system.two_machines;
```

Exploitation d'une description

- une description AADL doit contenir les informations nécessaires à son exploitation
  - ex.: les temps d'exécution sont inutiles pour vérifier la taille totale des composants logiciels
- outils spécialisés
  - vérifier la cohérence de l'architecture
    - *ex.: l'architecture est conforme aux contraintes des outils d'exploitation*
  - vérifier que les propriétés annoncées sont respectées
    - *ordonnançabilité, taille mémoire*
    - *spécifiées par les propriétés AADL*
  - générer ou simuler un système
    - *descriptions comportementales associées aux composants par des propriétés*
    - *code source : C, Ada, VHDL, …*
    - *ou descriptions formelles (p.ex. réseaux de Petri)*
  - etc.

Ocarina

- ensemble d'outils pour décrire et générer une application répartie
- éditeurs
  - texte : Emacs
  - graphique : Dia
- bibliothèque centrale
  - vérifications sémantiques générales
  - peut être utilisée par des applications existantes
    - *Cheddar (univ. Brest) : analyse d'ordonnancement*
    - *GLADE (AdaCore/ENST) : annexe des systèmes répartis pour Ada 95*
  - plusieurs modules de syntaxe
    - *texte*
    - *graphique (Dia)*
    - *XML*
- générateur de code : Gaia



Paramètres pour la génération du système

- composants
  - matériels : fournissent les informations de déploiement
    - *caractéristiques des machines, connexions sur les réseaux, etc.*
  - 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*
- besoin d'un exécutif AADL pour supporter les applications générées
  - symbolisé par les threads AADL
  - contrôle de l'exécution des threads système
  - prise en charge des communications
    - *fonction des interfaces des threads*
    - *passage de messages, sous-programmes distants, objets distribués, mémoire partagée*
- plusieurs façons d'aborder la génération de l'application
  - influe sur la modélisation AADL

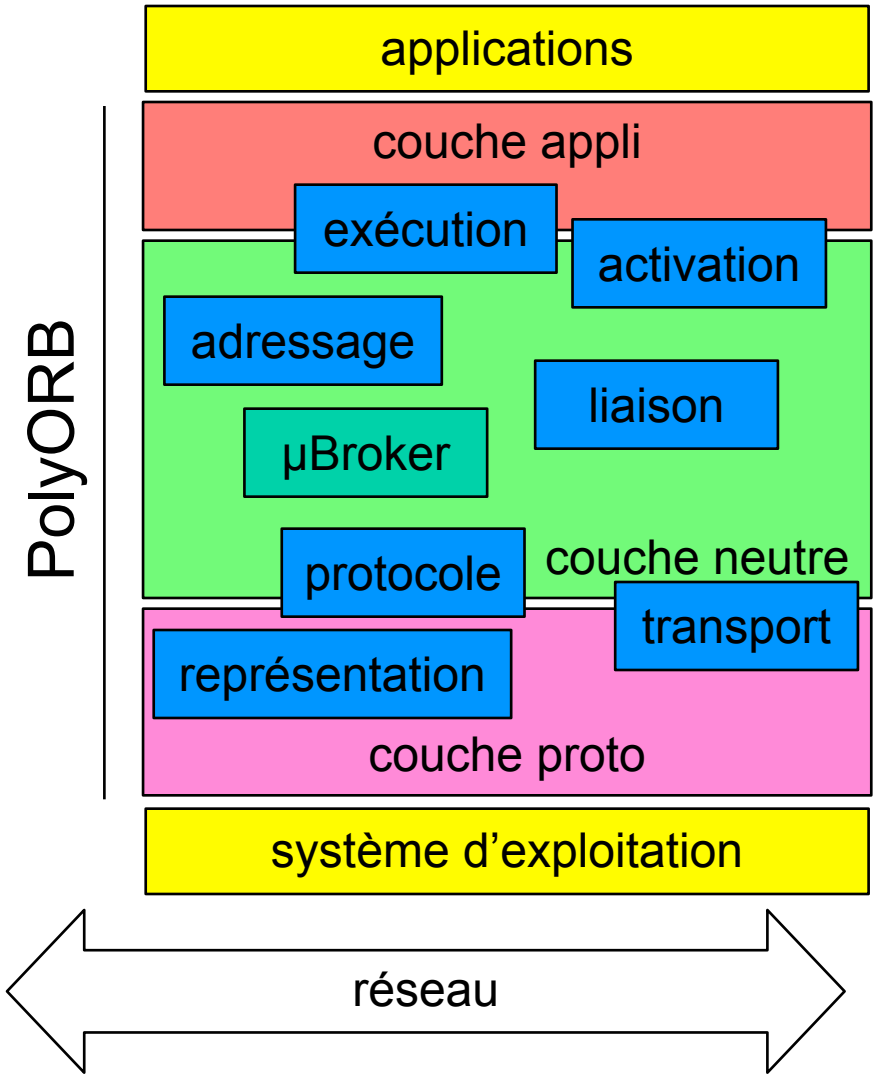
Déploiement automatique d'une application

- utilisation d'AADL pour décrire le placement des nœuds
  - processus
  - connexions
  - processeurs
  - propriétés
- modélisation gros grain
  - pas de sous-programmes, pas de threads
  - structure des applications laissée à l'utilisateur
  - peu de vérification possible

- Ex.: GLADE/AADL
  - implémentation de l'annexe des systèmes répartis d'Ada 95
  - pas de prise en compte de la modélisation des threads
  - exécutif AADL = exécutif de GLADE (Garlic) + exécutif Ada
  - traduction des éléments de configuration de GLADE en AADL

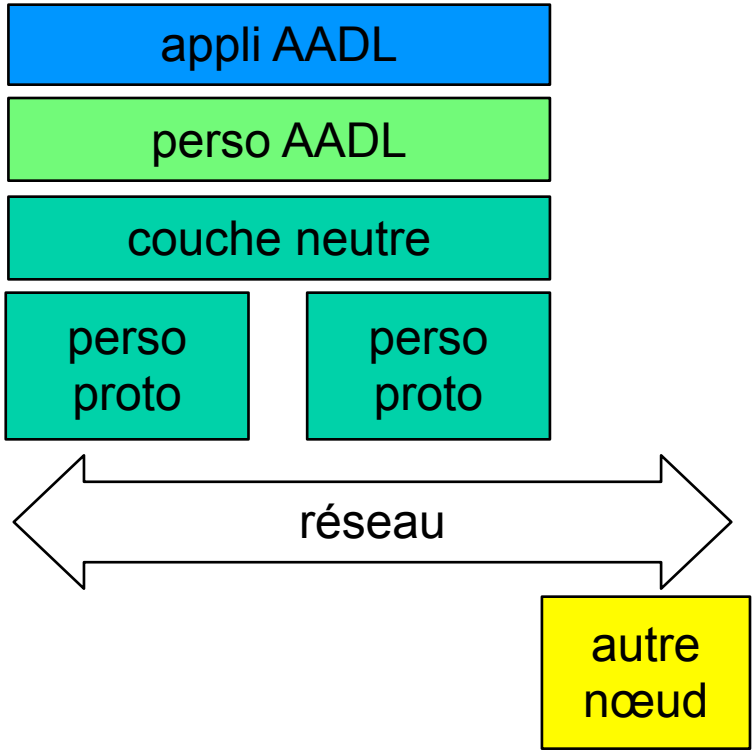
utilisation d'un intergiciel : PolyORB

- implémentation de l'architecture d'intergiciel schizophrène
- organisation en trois couches
  - personnalités applicatives : CORBA, DSA, MOMA, …
  - couche neutre : composants génériques d'intergiciel, indépendant des modèles de répartition
  - personnalités protocolaires : GIOP, SOAP, …
- architecture canonique
  - 7 services fondamentaux : adressage, liaison, représentation, protocole, transport, activation, exécution
  - un composant central de coordination : le µBroker
- adaptabilité
  - sélection & adaptation des composants
  - utilisation de personnalités adaptées aux applications et aux protocoles à utiliser
- fiabilité
  - µBroker modélisé en réseaux de Petri



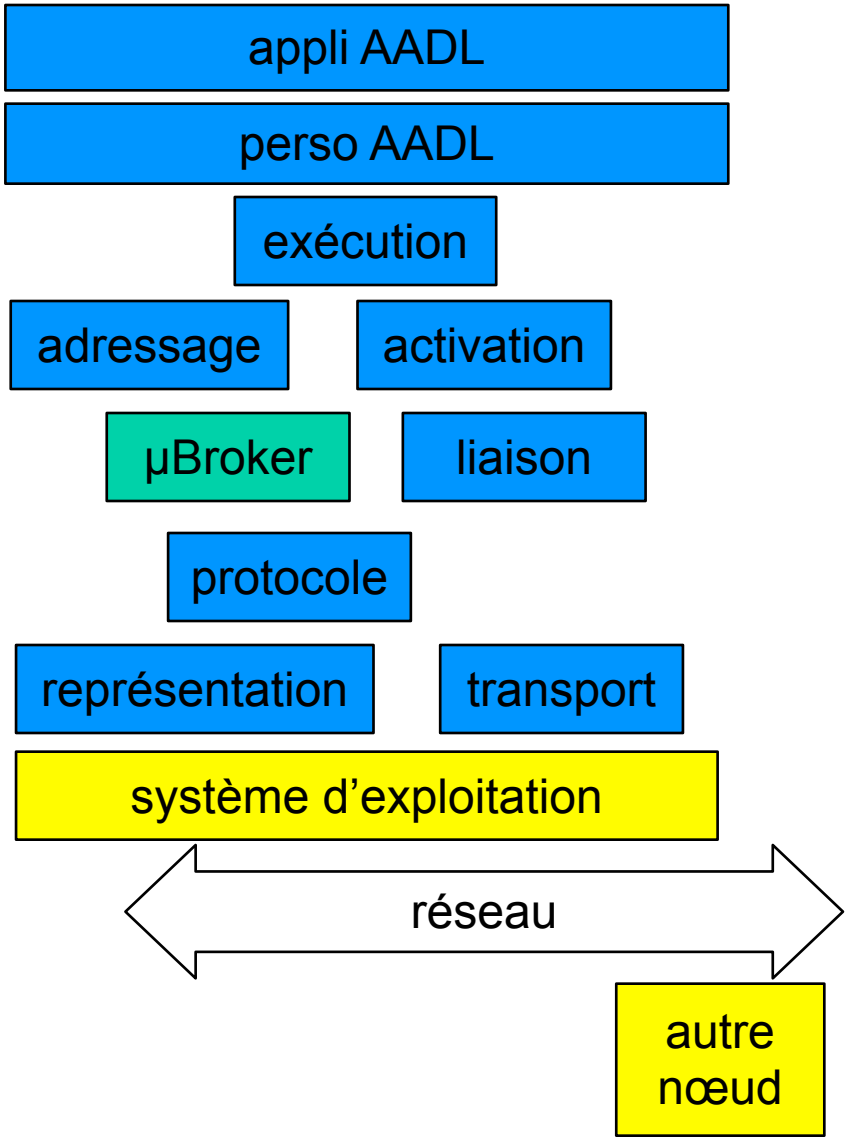
Génération complète reposant sur un intergiciel

- exécutif AADL = intergiciel
  - thread AADL
    - *thread fourni par l'intergiciel*
    - *ou thread système si l'intergiciel n'en offre pas*
  - communications assurées par l'intergiciel
- générateur de code en deux parties
  - génération des éléments applicatifs
    - *à partir des sous-programmes AADL*
  - génération d'une couche logicielle pour adapter l'intergiciel à l'application AADL
    - *à partir des threads AADL*
    - *traduction des différents modèles de communication en fonction de l'intergiciel*
- les composants applicatifs sont pilotés par l'exécutif
  - code utilisateur encapsulé, n'a aucun contrôle sur l'exécutif
  - intergiciel configuré en fonction des informations fournies par la description AADL
  - possibilité de vérification
    - *à condition de connaître les caractéristiques de l'intergiciel utilisé*
  - le contrôle par l'intergiciel impose certaines restrictions sur l'architecture
    - *modèles de conceptions architecturaux*
- ex.: Gaia/PolyORB

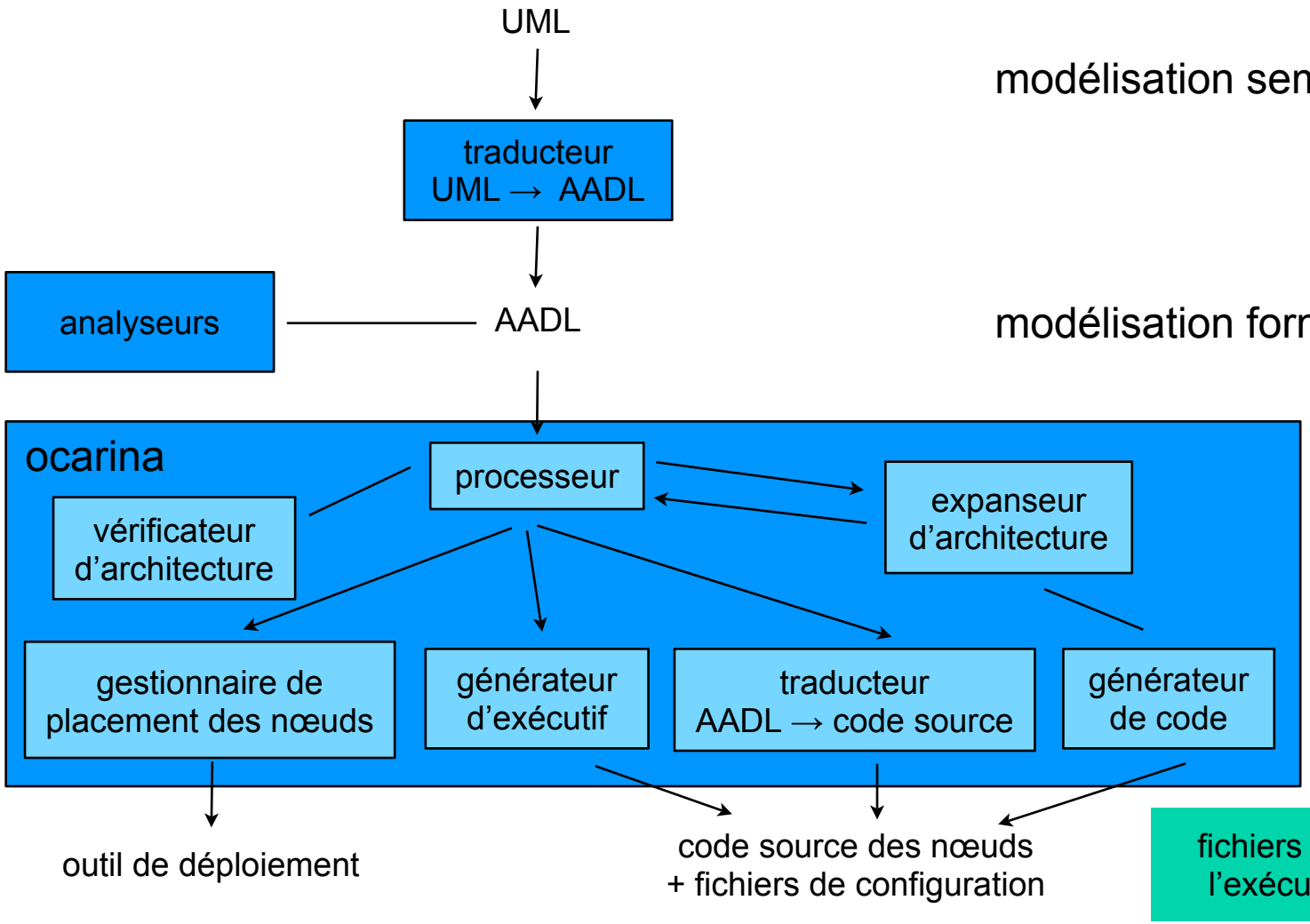


Génération des composants de l'intergiciel

- transformation des threads AADL
  - expansion des threads AADL pour introduire les services de l'architecture schizophrène
  - génération du code associé, ou association avec du code existant
- exécutif minimal et vérifiable
  - contrôle de l'exécution par le µBroker
    - *modélisé en réseaux de Petri*
  - communications à travers une bibliothèque de sockets
    - *dont l'interface est décrite en AADL*
    - *fournie par un système d'exploitation éventuellement modélisé en AADL*
- description quasi-complète par composants AADL assemblés
  - facilite l'analyse de l'application
  - interopérabilité inter-langage
  - intergiciel complètement générique
- ex.: Gaia/µBroker
  - exécutif AADL = µBroker de PolyORB
  - les composants de l'intergiciel schizophrène sont modélisés comme des éléments de l'application



Processus de conception



modélisation semi-formelle abstraite

modélisation formelle concrète

Conclusion

- AADL offre une grande souplesse de modélisation
  - degré de modélisation selon les besoins
    - *simple description*
    - *génération de code*
    - *simulation/analyse*
    - *…*
  - peut être utilisé comme langage fédérateur
    - *exploitation par plusieurs outils différents : générateur, analyseurs, visualiseurs, …*
- modélisation concrète
  - dernière phase avant la génération/déploiement du système
  - peut être utilisé dans une chaîne de modélisation
    - *conception avec des formalismes plus abstraites (UML, RdP, etc.)*
    - *génération d'une modélisation intermédiaire en AADL*
    - *analyse puis génération et déploiement de l'application*
- http://ocarina.enst.fr
- http://www.aadl.info
- http://polyorb.objectweb.org