Master 2 SETI

Spécialisation : Système et outils logiciels

UE B3 : Optimisation des systèmes distribués temps réel embarqués

2020-2021

Yves Sorel

Inria Paris
2 rue Simone Iff
CS 42112
75589 Paris Cedex 12
Tél. : 01 80 49 40 11, email : yves.sorel@inria.fr

http://www.syndex.org

Plan

Contexte et objectifs

Exemples de systèmes embarqués

Automobile

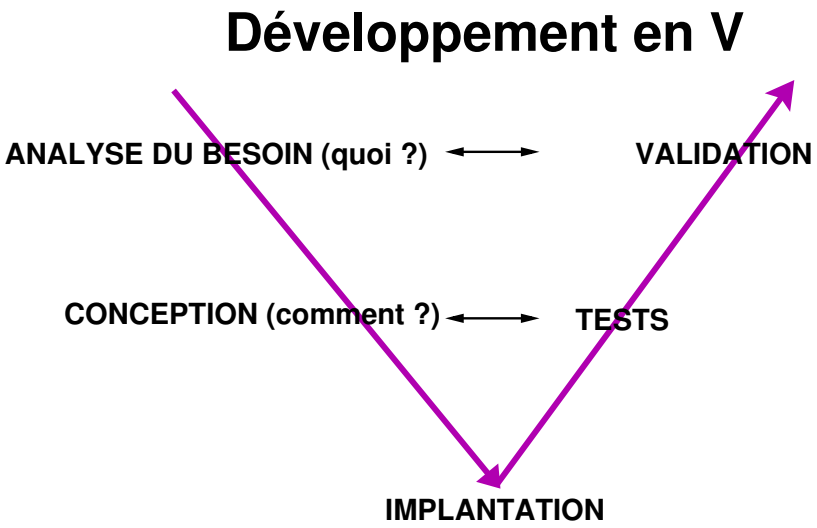
Avionique

Robotique mobile

Telecommunication

Approche système

Développement des systèmes embarqués

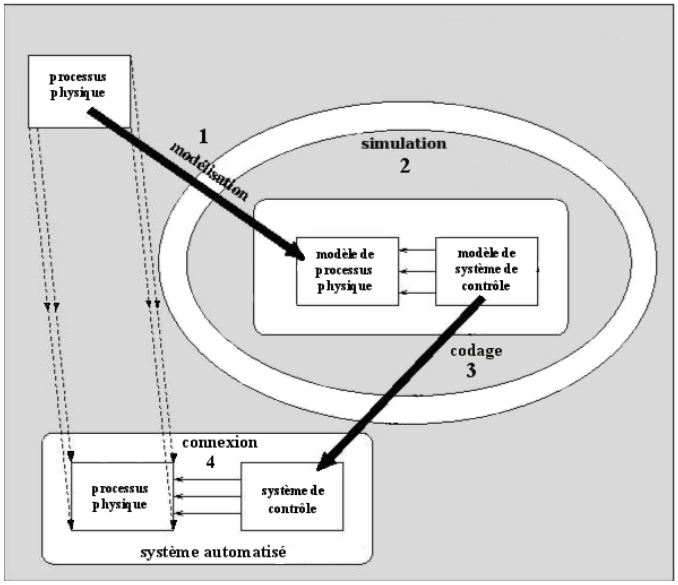


On traite les phases basses du **V** : **Conception** → **Implantation** → **Tests** en cherchant à minimiser les opérations manuelles, voire de les supprimer.

On vise un développement en **I sans remontée dans la partie droite du V (tests, validation) garantissant**, pour les phases **de conception et d'implantation, la sûreté de fonctionnement**, **par construction**.

Approche système

Liens entre AUTOMATIQUE et INFORMATIQUE



Un **système automatisé** est composé d'un **processus physique** connecté à un **système de contrôle** (**contrôleur**) numérique qui produit une **commande** conduisant le processus physique dans un état déterminé. Un **système cyber-physique** est un système automatisé ou un ensemble de systèmes automatisés **communiquant** à l'aide d'un réseau ad hoc ou de l'internet.

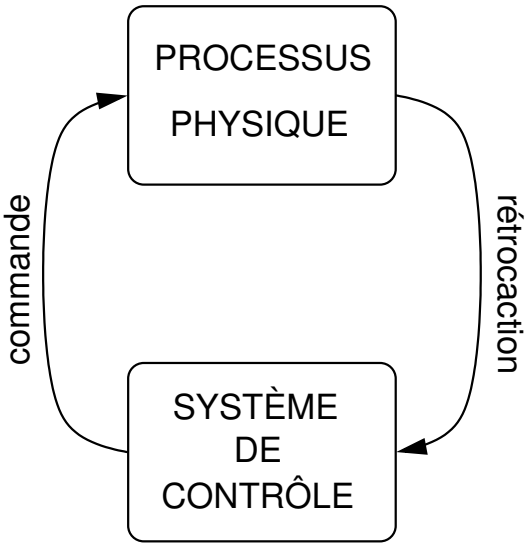
AUTOMATIQUE : CONCEPTION : (1) modélisation et (2) simulation hybride (processus physique continu et système de contrôle discret).

INFORMATIQUE : IMPLANTATION : (3) codage du système de contrôle sur processeurs et/ou circuits intégrés spécifiques (CIS) et (4) connexion avec le processus.

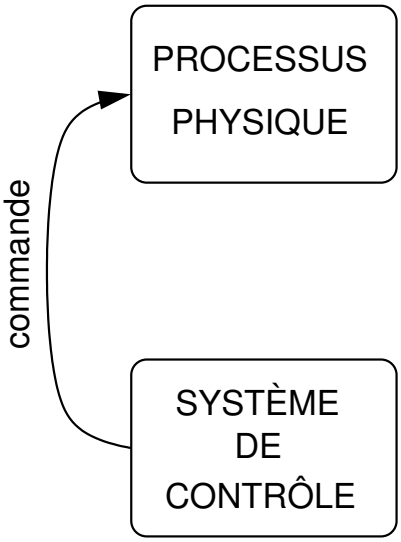
Approche système

Structure générale d'un système automatisé

Dans un système automatisé le système de contrôle est connecté au processus physique afin de le conduire dans un état déterminé. Le processus physique peut être connecté au système de contrôle afin de lui fournir son état. Tout cela s'étend aux systèmes cyber-physiques.



(a) boucle fermée



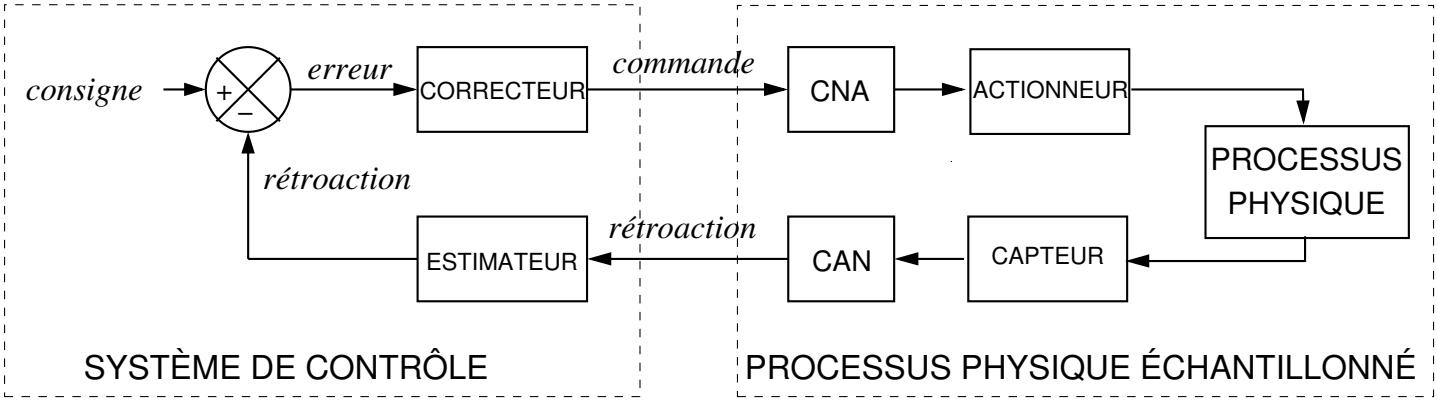
(b) boucle ouverte

Approche système

Système de contrôle bouclé sur processus physique échantillonné

La connexion entre le système de contrôle et le processus physique se fait à travers un **actionneur** associé à un **convertisseur numérique analogique** (CNA).

La connexion entre le processus physique et le système de contrôle se fait à travers un **capteur** associé à un **convertisseur analogique numérique** (CAN), conduisant à un **processus physique échantillonné**.



Définitions

Système réactif

Le système de contrôle produit une **infinité d'événements de sortie** qui sont des valeurs numériques consommées par le CNA connecté au processus physique. Le système de contrôle consomme une **infinité d'événements d'entrée** qui sont des valeurs numériques produites par le CAN connecté au processus physique. Chaque suite infinie d'événements **valués** d'entrée ou de sortie est appelée un **signal**.

Evénements d'entrée $\Rightarrow$ SYSTÈME DE CONTRÔLE RÉACTIF ET TEMPS RÉEL $\Rightarrow$ Evénements de sortie

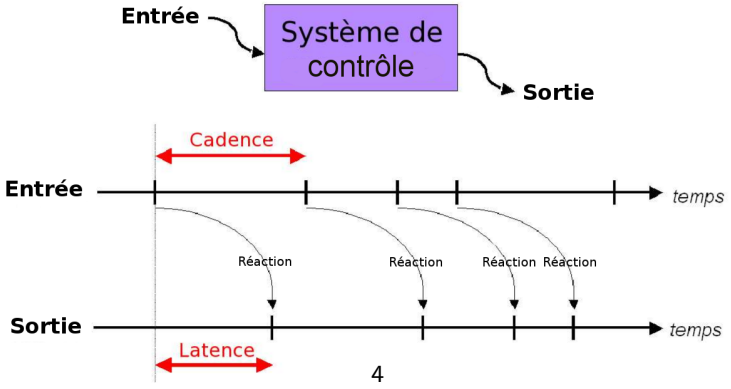
Pour qu'un système de contrôle soit **réactif** (Harel, Pnueli 1985), il **faut que** lorsqu'il consomme un **événement d'entrée** il produise, **en réaction**, un **événement de sortie**. Chaque événement d'entrée est consommé par un ensemble de fonctions qui produit un événement de sortie correspondant à l'événement d'entrée. Cet ensemble de fonctions se répète donc, de manière infinie, à chaque entrée d'événement.

Définitions

Système temps réel

Un **système temps réel doit** être réactif et respecter des contraintes temporelles de deux types :

- **cadence** : contrainte sur la durée s'écoulant entre deux événements d'un même signal d'entrée qui peut être périodique, sporadique avec période minimale, ou apériodique sans période ;
- **latence entrée-sortie** : contrainte sur la durée s'écoulant entre l'arrivée d'un événement d'un signal d'entrée et la production de l'événement du signal créé, **en réaction**, par un ensemble de fonctions du système de contrôle.



Définitions

Système temps réel distribué embarqué, déclenché par événements ou temps

Système temps réel critique, strict, dur : le non respect des contraintes a des conséquences catastrophiques, pertes humaines, écologiques, etc.

Système temps réel souple, mou, QoS : on accepte qu'un certain pourcentage des contraintes ne soit pas respecté : qualité de service.

Système distribué, parallèle, multiprocesseur, multi(pluri)cœur : pour performances, modularité, rapprocher les calculs des capteurs/actionneurs.

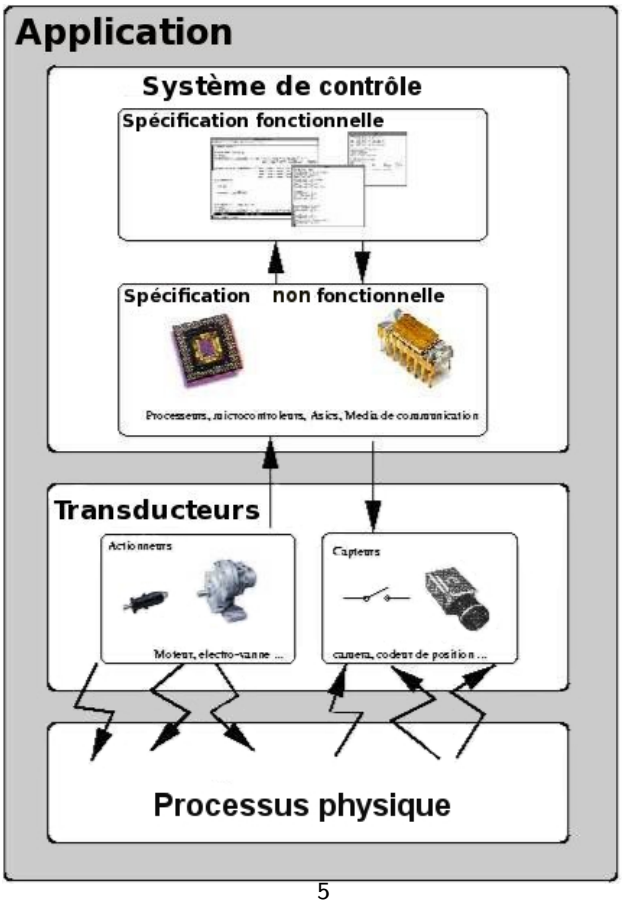
Système embarqué : demande une minimisation des ressources (encombrement, poids, consommation, coût, etc.).

Système déclenché par des événements (Event Triggered ET) : l'état du processus physique est connu via des interruptions produites par les capteurs, les actionneurs suivent, flexible, **adapté au temps réel souple**.

Système déclenché par le temps (Time Triggered TT) : l'état du processus physique est connu par interrogations des capteurs à l'aide d'un compteur de temps discret, les actionneurs sont synchronisés, rigide, **adapté au temps réel critique**.

Définitions

Application temps réel : système automatise' sans processus physique



Domaines d'application

Systèmes grand public et à large échelle

- **Systèmes grand public**
  - télécoms : téléphone mobile intelligent, modems, routeurs, etc.
  - automobile : contrôle moteur, ABS, ESP, conduite aidée/autonome,etc.
  - robotique : robot humanoïde et industriel, exosquelette, drones, etc.
  - médecine : suivi patient capteurs EEG ECG, injection automatique, etc.
  - domotique : télésurveillance, automatisation tâches ménagères, etc.
  - équipements audio et vidéo : baladeur, set-up box, télévision HD, etc.
  - objets connectés : montre, bracelet de santé, pèse personne, etc.

On vise la minimisation du coût

- **Systèmes à large échelle**
  - aéronautique et spatial
  - contrôle du trafic aérien
  - transports terrestres, ferroviaire, bus, métro
  - contrôle de processus industriels
  - centraux télécoms, stations de base
  - systèmes d'armes

On vise la rapidité du développement

Spécification fonctionnelle

Conception : spécification du système de contrôle : automatique

Les **automaticiens** spécifient les fonctions et les dépendances de donnée entre les signaux de sortie des fonctions **productrices** de données et les signaux d'entrée des fonctions **consommatrices** de données. Les fonctions peuvent être de type traitement du signal et des images ou de type contrôle/commande.

Traitement du signal et des images $\Rightarrow$ Calculs nombreux
Algorithmes réguliers - `For i=1 to N Do`
Implantation plus simple

Contrôle/commande $\Rightarrow$ Calculs peu nombreux
Non réguliers - `If cond Then Else`
`Changement de modes`
Implantation plus complexe

De manière générale on a un mélange d'algorithmes réguliers et non réguliers, ce qui complique le problème d'implantation.

Spécification non fonctionnelle

Conception : spécification de l'architecture matérielle : informatique

Les **informaticiens** spécifient les composants de l'architecture matérielle et la manière dont ils sont interconnectés, et les contraintes de distribution et d'ordonnancement sur les fonctions relativement aux composants, et sur les dépendances de donnée relativement aux média de communication.

$$\frac{\text{Durée de toutes les fonctions}}{\text{Contrainte de latence}} > 1 \Rightarrow \text{Architecture distribuée, parallèle, etc.}$$

Architecture hétérogène dite **multicomposant** (Lavarenne, Sorel 96) formée de :

- **capteur et actionneur** ;
- **composant programmable** : processeurs RISC, CISC, DSP (Digital Signal Processor), ASIP (Application Specific Instruction set Processor) ;
- **composant non programmable** : carte spécialisée, CIS : ASIC (Application Specific Integrated circuit) et FPGA (Field Programmable Gate Array) ;
- **médium de communication** : liaison point-à-point (crossbar), liaison multi-point (bus), réseau, etc.

Spécification non fonctionnelle

Conception : spécification des caractéristiques temporelles : automatique et informatique

Il faut aussi spécifier les **caractéristiques temporelles** associées à chaque fonction de la spécification fonctionnelle.

Elles sont spécifiées par les :

- **automaticiens** et sont **indépendantes de l'architecture** : période, période minimale dans le cas sporadique, contrainte d'échéance, contrainte de latence généralisée sur couple de fonctions pas nécessairement entrée-sortie ;
- **informaticiens** et sont **dépendantes de l'architecture** : pire temps d'exécution WCET (Worst Case Execution Time) des fonctions sur les composants et pire temps d'exécution des communications inter-processeur dues aux dépendances de donnée WCCT (Worst Case Communication Time) sur les média de communication.

Les spécifications de propriétés de **sûreté de fonctionnement sont garanties par construction**.

Les spécification de propriétés de **tolérance aux fautes** et de **cyber sécurité ne sont pas traitées dans le cours**.

Implantation optimisée : méthodologie AAA

Parallélisme potentiel vs parallélisme effectif 1/3

L'**implantation** consiste à choisir sur quel processeur et/ou sur quel CIS de l'architecture parallèle chaque fonction sera distribuée et, pour chaque processeur séquentiel dans quel ordre elle y sera exécutée.

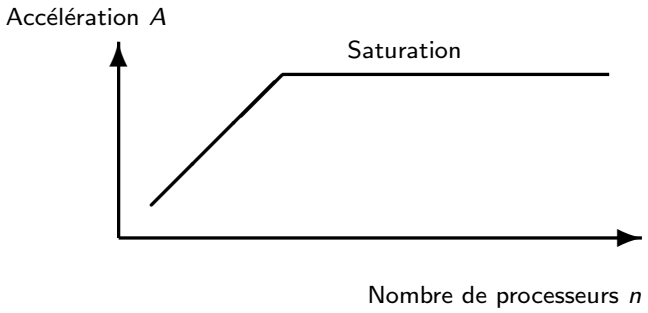
La **méthodologie AAA** qui vise à **optimiser** l'implantation s'appuie sur des notions importantes présentées dans la suite : **parallélisme** et **système d'exploitation**.

Le **parallélisme potentiel de contrôle** (fonctions différentes appliquées à des données différentes) et le **parallélisme potentiel de données** (même fonction appliquée à des données différentes) associés à une spécification fonctionnelle et appelés tous les deux **parallélisme potentiel** par la suite, correspondent à l'ensemble des fonctions qui ne sont pas dépendantes car, dans ce cas, elles pourront s'exécuter potentiellement en parallèle. Le parallélisme potentiel ne **dépend pas** des durées d'exécution des fonctions.

Implantation optimisée : méthodologie AAA

Parallélisme potentiel vs parallélisme effectif 2/3

Quand le **parallélisme effectif** de l'architecture est inférieur ou égal à celui du **parallélisme potentiel**, on peut avoir une **accélération** du temps d'exécution qui est proportionnelle au nombre de processeurs. Dès qu'il lui est supérieur il y a **saturation** de l'accélération qui n'augmente plus avec le nombre de processeurs.



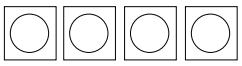
L'accélération $A = \frac{T_{monoprocesseur}}{T_{multiprocesseur}}$. Le phénomène de saturation est formalisé dans la loi d'Amdahl qui dit que $A = \frac{1}{(1-P)+\frac{P}{n}}$ est non linéaire en n, avec P = pourcentage du programme parallélisable ou **parallélisme potentiel**.

Implantation optimisée : méthodologie AAA

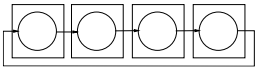
Parallélisme potentiel vs parallélisme effectif 3/3

Par ailleurs, des fonctions dépendantes peuvent s'exécuter en **parallélisme potentiel de flot (pipeline) si elles se répètent**, comme c'est le cas dans les systèmes réactifs, mais l'accélération obtenue qui est inférieure à celle obtenue en parallélisme potentiel, est de plus diminuée des temps de communications dûs aux dépendances.

Si on suppose que les fonctions ne sont pas dépendantes et que l'on a n fonctions distribuées sur n processeurs et que chaque fonction a une durée de 1, le parallélisme potentiel de contrôle et de données donne une accélération en latence $A = n/1 = n$, pour $n = 4$ on a $A = 4$.



Avec une chaîne de n fonctions dépendantes distribuées sur n processeurs, si cette chaîne se répète et si on suppose que le coût des communications est nul, le parallélisme potentiel de flot donne une accélération en latence $A = \frac{n^2}{(2n-1)}$ alors qu'on a une accélération en cadence (débit) de n pour un nombre suffisamment grand de répétitions. Pour $n = 4$ on a $A = 2,3$.



Implantation optimisée : méthodologie AAA

Système d'exploitation : fournit aux fonctions des services

Algorithme

Système d'exploitation

Architecture

Algorithme : définition informelle (Al-Khwarizmi, astronome Perse 825) description d'une fonction ou d'un ensemble de fonctions, dépendantes ou pas, à l'aide un nombre fini d'instructions choisies dans un ensemble fini d'instructions, définition formelle (Turing, Post 1930).

Système d'exploitation (OS Operating System) : fournit aux fonctions des services pour exploiter les composants programmables (processeur) : programmes, mémoire, fichiers, périphériques, interruptions, ordonnancement, communications.

Système d'exploitation temps réel (RTOS Real-Time OS) : fournit ces services en respectant des contraintes temps réel. Il est appelé dans la suite **exécutif** quand on fait référence seulement aux services de bas niveau : interruptions, mémoire, ordonnancement, communications, synchronisations.

Architecture : électronique numérique constituée de capteurs et d'actionneurs, de processeurs et/ou CIS, de média de communication.

Implantation optimisée : méthodologie AAA

Exécutif : allocation des ressources de calcul, mémoire et communication

Fonctionnalités de tous les OS

OS = Allocation spatiale et temporelle des ressources matérielles de types **calcul**, **mémoire**, **communication** à l'algorithme

+

Distribué → Plusieurs ressources matérielles pour chaque type

+

Fonctionnalités des RTOS

Réactif → Ordre événements de sortie = ordre événements d'entrée quel que soit le temps d'exécution de la réaction

+

Temps réel → Allocation déterminée par l'écoulement du temps physique et le respect des contraintes temps réel

Implantation optimisée : méthodologie AAA

Exécutif : allocation de ressources spatiale vs temporelle, hors ligne vs en ligne

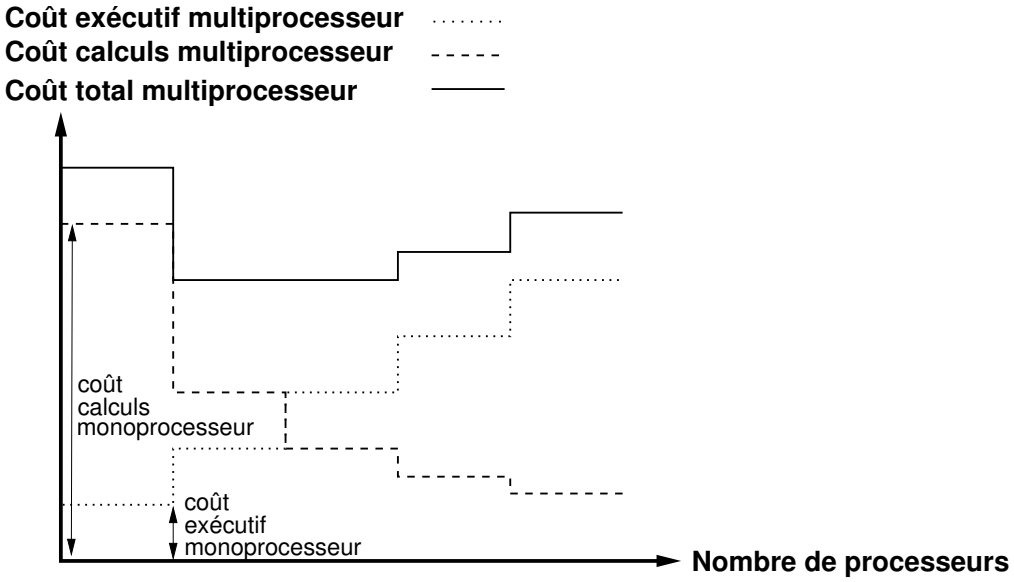
L'exécutif temps réel alloue des ressources

- Distribution : allocation spatiale
- Ordonnancement : allocation temporelle
- En ligne : choix d'allocations et d'optimisations effectués **pendant** l'exécution, fondé sur le traitement d'interruptions, **non déterministe** car coût de l'exécutif variable
- **Déterministe** veut dire que si on exécute plusieurs fois le système de contrôle avec les mêmes entrées on obtient les mêmes sorties
- Hors ligne : choix d'allocation et d'optimisation effectués **avant** l'exécution, fondé sur l'utilisation d'une table d'ordonnancement, **déterministe** car coût de l'exécutif constant

Implantation optimisée : méthodologie AAA

Exécutif : il a un coût

L'exécutif temps réel a un coût qui augmente avec le nombre de processeurs à cause des **communications et synchronisations inter-processeurs** (dues aux dépendances de donnée) qui sont de plus en plus nombreuses, alors que le coût des calculs, lui, diminue.



$$cout_{Total} = cout_{Executif} * n + cout_{Calculs}/n$$

Implantation optimisée : méthodologie AAA

Objectifs

A partir d'une **spécification fonctionnelle** et d'une **spécification non fonctionnelle** (architecture matérielle et caractéristiques temporelles) réalisées lors de la **conception**, il faut explorer les **implantations** possibles (allocation spatiale et temporelle des fonctions aux ressources matérielles considérées comme des machines séquentielles) pour obtenir, soit manuellement ou soit automatiquement, une **implantation optimisée sûre par construction**.

L'exploration automatique s'effectue sur des **modèles formels basiques** représentant l'algorithme et l'architecture (graphes, ordre partiel, machine à états finie) en réalisant des transformations de graphes fondées sur des **analyses d'ordonnançabilité temps réel distribué** et des **optimisations temporelles et de ressources**.

Implantation optimisée : méthodologie AAA

Modèles formels basiques

Un **graphe** G est un couple (S, A) où S est un ensemble fini de sommets et A est une relation binaire sur S définissant des couples de sommets $(s_1, s_2) \in S \times S$ tels que $(s_1\, A\, s_2) \Leftrightarrow (s_1 \ll \text{est relié à} \gg s_2)$ par une **arête**. Il est dit **orienté** si chaque couple (arête) est ordonné $(s_1, s_2) \neq (s_2, s_1)$, on a alors $(s_1 \ll \text{précède} \gg s_2)$. Un couple ordonné est appelé **arc**. Un **chemin** (resp. **chaîne**) est une suite finie d'arcs (resp. d'arêtes) consécutifs. Un graphe est **acyclique** (resp. sans boucle) s'il n'a pas de chemin (resp. de chaîne) dont le dernier sommet est aussi le premier sommet, ex. : $(s_1, s_2), (s_2, s_3), (s_3, s_1)$.

Pour un **graphe orienté** G, la relation A est antisymétrique, c.a.d. que si $s_1 A s_2$ et $s_2 A s_1$ alors $s_2 = s_1$, transitive, c.a.d. que si $s_1 A s_2$ et $s_2 A s_3$ alors $s_1 A s_3$ et non réflexive, c.a.d. qu'elle ne concerne que des éléments distincts $s \neg A s$, c'est donc une **relation d'ordre strict**, notée $<$. Si elle est réflexive c'est une **relation d'ordre non strict**, notée $\leq$. A est une **relation d'ordre strict partiel** si $A \subset S \times S$, c'est une **relation d'ordre strict total** si $A = S \times S$. Un chemin forme un ordre total, une chaîne ne forme pas un ordre total.

Pour un **graphe orienté** G, un **stable** est un sous-ensemble de sommets qui ne sont pas deux à deux en relation A, c.a.d. qui ne sont pas deux à deux reliés par un arc ou plusieurs arcs par transitivité. Le **plus grand stable** du graphe est noté A^*. Si $A \subset S \times S$ on a $A \cup A^* = \bar{A}$ avec $\bar{A} = S \times S$.

Pour un **graphe non orienté** G, la relation A est symétrique, c.a.d. que si $s_1 A s_2$ alors $s_2 A s_1$, transitive et réflexive, c'est donc une **relation d'équivalence**.

Implantation optimisée : méthodologie AAA

Principes

ADEQUATION

ALGORITHME	**Caractéristiques temporelles**	**ARCHITECTURE**
↓		↓
graphe algorithme	→ *transformations* ←	graphe architecture
↓		↓
parallélisme potentiel	← *réduction* ←	parallélisme effectif

OPTIMISATIONS

↙ ↘

Processeurs
EXECUTIF
- hors ligne, synthétisé sur mesure
- peut appeler un exécutif résident en ligne : VxWorks, Osek, Linux/RTAI, Linux/Xenomai, Windows/RTX, etc.

CIS
NETLIST

Spécification des Algorithmes

Généralités

Algorithme : fonction ou ensemble de fonctions modélisé par un graphe

La spécification fonctionnelle décrit le système de contrôle sous la forme d'un **algorithme** qui devra être **implanté** en utilisant les processeurs et/ou les CIS de l'architecture et les caractéristiques temporelles donnés lors de la spécification non fonctionnelle.

L'algorithme décrit, éventuellement de manière **hiérarchique**, les fonctions ainsi que l'ordre partiel maximal (parallélisme potentiel) dans lequel ces fonctions devront s'exécuter à cause de leurs dépendances de donnée. Il décrit aussi comment certaines fonctions seront **exécutées conditionnellement** ou seront **répétées infiniment** ou bien **un nombre fini de fois**.

Il y a deux approches principales pour décrire un algorithme à l'aide de modèles formels fondés sur des graphes : **graphe flot de contrôle** ou **graphe flot de données**.

Modèle d'algorithme

Graphes flot de contrôle 1/3

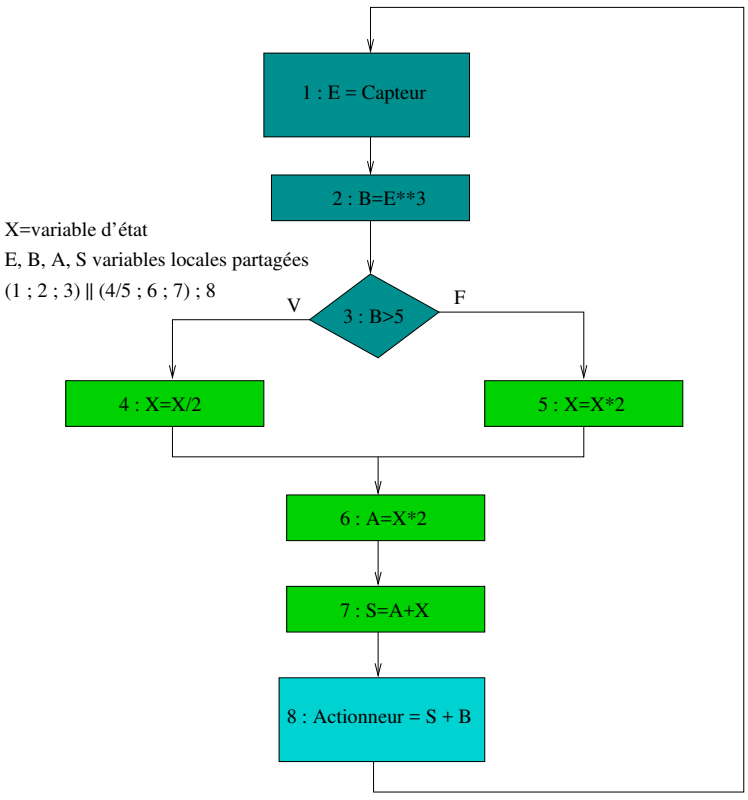
L'algorithme peut être modélisé par un graphe orienté cyclique appelé **graphe flot de contrôle** qui peut être de deux types :

- un **organigramme** dont les **sommets** sont des fonctions que l'on appellera dans la suite **opérations** pour correspondre à la notion d'**opérateur** du modèle d'architecture, et les **arcs orientés** des **dépendances de contrôle** (branchement avant inconditionnel - exécution en séquence d'opérations - et branchement avant conditionnel sur test pour exécution alternative d'opérations appelé divergence en `OU`, branchement arrière pour boucle) qui induisent des précédences entre opérations. Une opération correspond à une **séquence d'instructions d'un processeur**, elle est exécutée dès que l'opération dont elle dépend est terminée, elle lit et écrit ses données dans des variables locales ou d'état ;
- un **automate** dont les **sommets** sont des états et les **arcs** orientés sont des transitions d'états. Une transition est réalisée (tirée) lors de l'arrivée d'un événement d'entrée induisant l'exécution d'une opération qui lit et écrit ses données dans des variables locales.

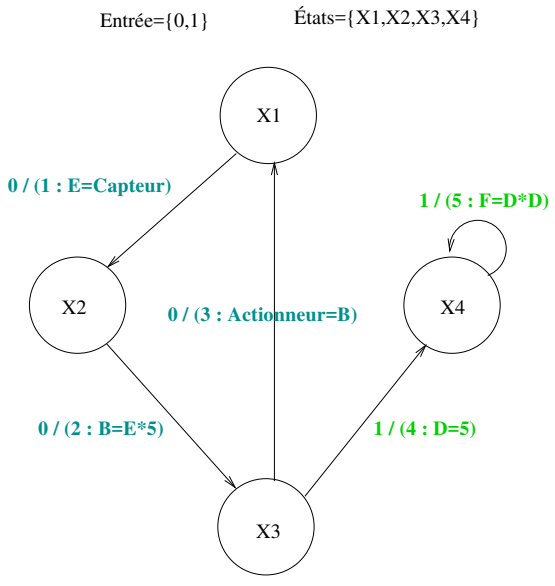
L'interaction avec le processus physique (système réactif) est modélisée par une boucle (organigramme) ou par l'arrivée des événements (automate).

Modèle d'algorithme

Graphes flot de contrôle 2/3



ORGANIGRAMME



D, B variables locales partagées

(1 ; 2 ; 3) || (4 ; 5)

AUTOMATE

Modèle d'algorithme

Graphes flot de contrôle 3/3

Dans ce modèle :

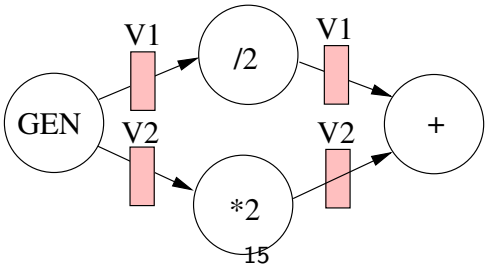
- les opérations accèdent aux données contenues dans des variables qui peuvent être **réutilisées et/ou partagées** par plusieurs opérations, ce qui est une source importante d'**erreurs** ;
- l'ordre dans lequel les opérations lisent et écrivent les données et l'ordre dans lequel ces opérations sont exécutées, ne sont pas liés ;
- toutes les opérations sont en relation de précédence, le flot de contrôle induit un **ordre total** sur l'exécution des opérations, pas de divergence en `ET` ;
- ce modèle introduit un **cycle de précédences** qui rend difficile la cohérence des accès en lecture/écriture aux variables.

Dans un organigramme la mémoire d'état est **implicitement localisée** dans les variables alors que dans le modèle d'automate elle est **explicitement localisée** dans les sommets.

Modèle d'algorithme

Graphe flot de données 1/3

L'algorithme peut être modélisé par un graphe orienté acyclique appelé **graphe flot de données** (Dennis, Kahn 1974) dont les sommets sont des opérations et les arcs orientés des **dépendances de contrôle et de données**. Ces dernieres définissent une **relation de précédence et de dépendance de donnée** entre opérations, appelée simplement **relation de dépendance** par la suite, à laquelle est associée une seule variable (**assignation unique**). Une opération est exécutée dès que toutes les données - produites par des opérations qui la précèdent - sont présentes sur ses **entrées**, elle produit alors toutes les données sur ses **sorties** - consommées par des opérations qui la succèdent (règle d'activation de Milner). Les données se **propagent (flot)** ainsi, depuis les sommets sans prédécesseurs jusqu'à ceux sans successeurs.



Modèle d'algorithme

Graphe flot de données 2/3

Dans ce modèle :

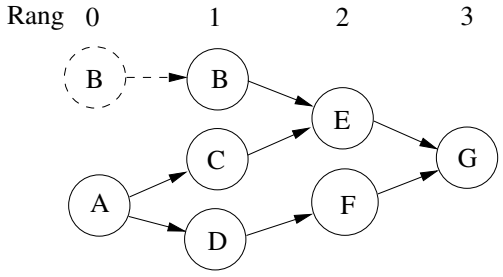
- à chaque arc correspond un transfert de donnée auquel est associé une seule variable, partagée uniquement par les deux opérations dépendantes. Les autres opérations ne peuvent pas y accéder. Le partage et la réutilisation globale de ces variables sera géré **automatiquement**, sans risque d'erreurs, par le compilateur plutôt que par l'utilisateur ;
- l'ordre dans lequel les données sont lues et écrites par les opérations, est cohérent avec l'ordre d'exécution des opérations qui les utilisent, cela évite que l'utilisateur impose un ordre sur les opérations et utilise les variables dans un ordre différent conduisant à une erreur ;
- certaines opérations **peuvent ne pas être en relation** de dépendance, cela induit un **ordre partiel** sur l'exécution des opérations : divergence en `ET` ;

Modèle d'algorithme

Graphe flot de données 3/3

- à partir des opérations qui ne sont pas en relation de dépendance, on définit le **parallélisme potentiel** inhérent à la spécification de l'algorithme. Plus formellement, c'est le **plus grand stable du graphe** sans considérer les sommets qui sont en relation d'ordre par transitivité. Pour le déterminer, on recherche tous les sous-ensembles de sommets qui ont le **même rang**, c.a.d. qui sont à la même distance, en nombre maximum de prédécesseurs, des sommets sans prédécesseurs. On peut déplacer un sommet pour augmenter le nombre de sommets d'un autre rang. Ici on peut déplacer B du rang 0 au rang 1, ce qui donne les stables $\{B, C, D\}$ et $\{E, F\}$.

  Le **parallélisme potentiel** correspondant au nombre de processeurs identiques que l'on peut mettre **potentiellement en parallèle**, est égal au cardinal du plus grand stable, ici $\{B, C, D\}$, avec $Card\{B, C, D\} = 3$;



- un sommet est **hiérarchique** s'il peut se décomposer en un graphe flot de données, sinon il est **atomique**. Un sommet atomique ne pourra pas être alloué (distribué) sur plusieurs ressources matérielles.

Modèle d'algorithme

Modèle AAA : graphe flot de données répété conditionné factorisé 1/3

Le modèle AAA est un **graphe flot de données répété conditionné factorisé**, c.a.d. un graphe flot de données classique étendu comme suit :

- **répété de manière infinie** : chaque répétition correspond à une interaction avec le processus physique (système réactif) et elle définit un instant `t` d'un **temps logique** d'un **système LTT** (Logical Time Triggered), les sommets sans prédécesseurs correspondent à des **capteurs** et les sommets sans successeurs à des **actionneurs** ;
- **conditionné** : certains sommets hiérarchiques se décomposent en plusieurs sommets, éventuellement hiérarchiques, dont un seul (divergence en `OU`) sera exécuté lors d'une répétition infinie en fonction de la valeur de son **entrée de conditionnement** respective qui recoit une **dépendance de conditionnement**. C'est une extension du graphe flot de données dynamique (Buck 1993). Un sommet conditionné correspond à un branchement conditionnel, `If...Then...Else...` dans le modèle flot de contrôle ;

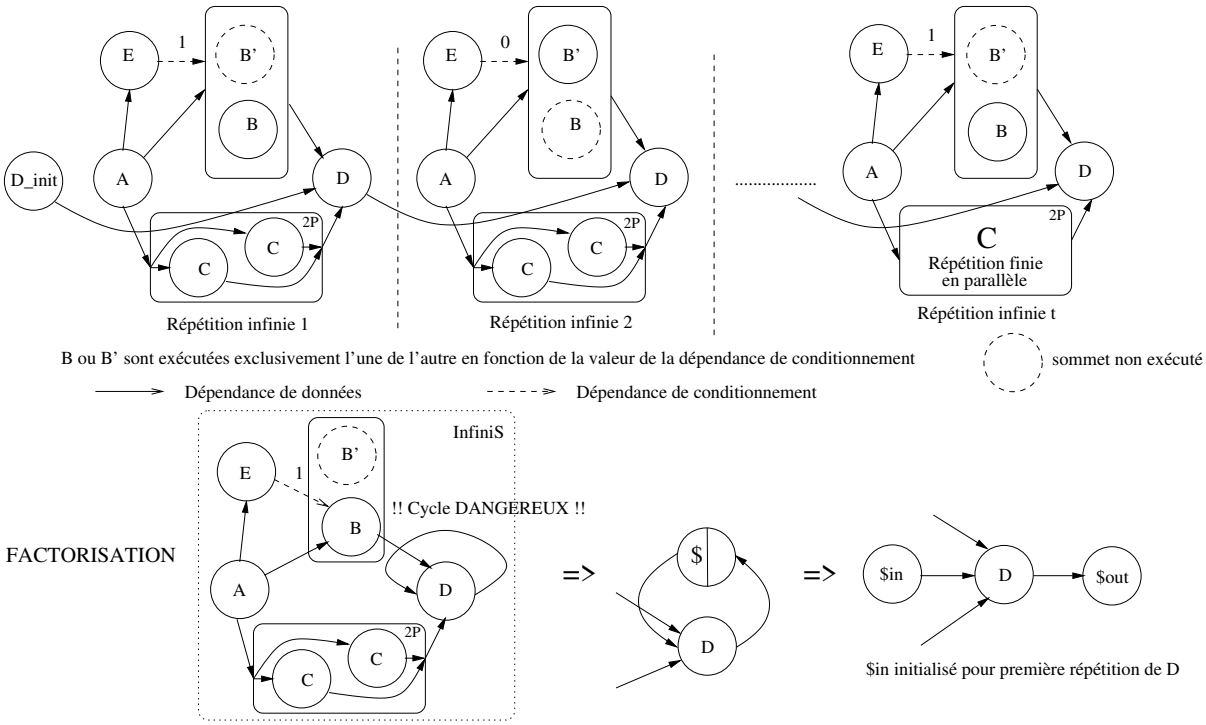
Modèle d'algorithme

Modèle AAA : graphe flot de données répété conditionné factorisé 2/3

- **répété de manière finie** : certains sommets hiérarchiques se décomposent en plusieurs sommets identiques qui seront tous (divergence en `ET`) exécutés sur des données différentes, en **parallélisme potentiel de données** ou **de flot**, par opposition au **parallélisme potentiel de contrôle** où les sommets sont différents. Un tel sommet possède un facteur de répétition nP en parallélisme de données quand cette opération s'applique à chacune des n données sans dépendances entre les opérations, ou un facteur de répétition nS en parallélisme de flot quand les n opérations, mises en série, forment un pipeline. Un tel sommet correspond à une boucle `For i=1 To N Do` dans le modèle flot de contrôle ;
- **factorisé** : afin de simplifier le modèle, mais cela fait apparaître des **cycles de dépendances de donnée** quand une opération de la répétition infinie t consomme des données produites à la répétition infinie $t - n$. Ces cycles sont **dangereux** car on ne sait pas si une donnée a été produite avant d'être consommée, créant ainsi un interblocage (deadlock). Chacun de ces cycles **doit** donc contenir au moins un **sommet retard ($)** définissant les **états** de l'algorithme qui **doivent** avoir une valeur initiale, celle consommée lors de sa première répétition infinie par une opération connectée à un retard.

Modèle d'algorithme

Modèle AAA : graphe flot de données répété conditionné factorisé 3/3

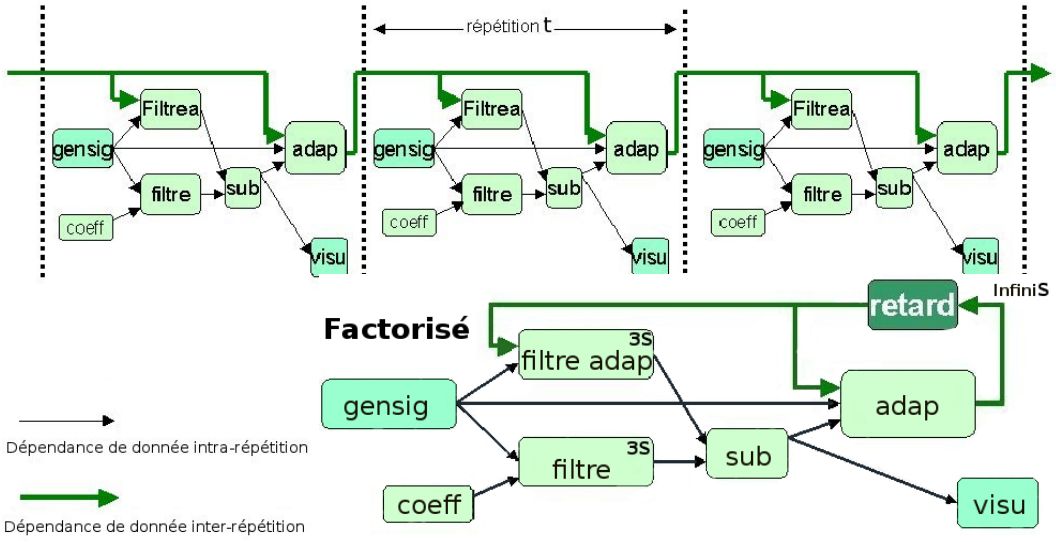


Chaque répétition infinie correspond à un instant logique t d'un temps logique.

Modèle d'algorithme

Exemple : égaliseur adaptatif

La sortie du capteur `gensig` passe par un **filtre transversal** `filtre` à coefficients fixes représentant le canal de transmission et par un autre filtre transversal `filtrea` qui reçoit ses coefficients calculés par la **fonction d'adaptation** `adap`. Cette dernière reçoit elle aussi les coefficients, la sortie du capteur et la différence des sorties des deux filtres formant une **erreur** visualisant avec l'actionneur `visu` la convergence de l'algorithme.



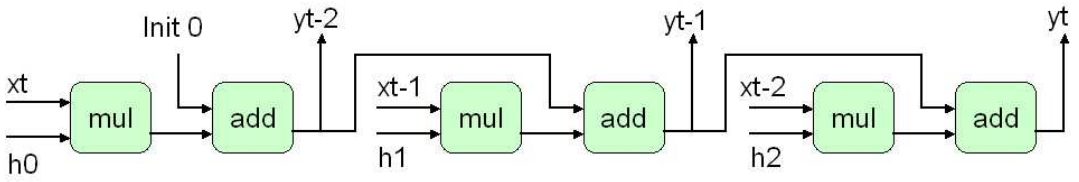
Modèle d'algorithme

Exemple : filtre transversal et algorithme d'adaptation

Filtre transversal : `filtre` **et** `filtrea`

Une entrée vecteur de 3 éléments contenant les coefficients $h_t = (h_0, h_1, h_2)$, une entrée vecteur de 3 éléments contenant le passé de la sortie du capteur (x_t, x_{t-1}, x_{t-2}), une sortie scalaire $y_t = \sum_{i=0}^{2} h_i x_{t-i}$

On répète 3 fois en série (3S) le motif (*mul*, *add*) tel que la sortie d'un *add* soit connectée avec l'une des entrées du *add* suivant.



Fork : H(h0, h1, h2) et X(xt, xt-1, xt-2)

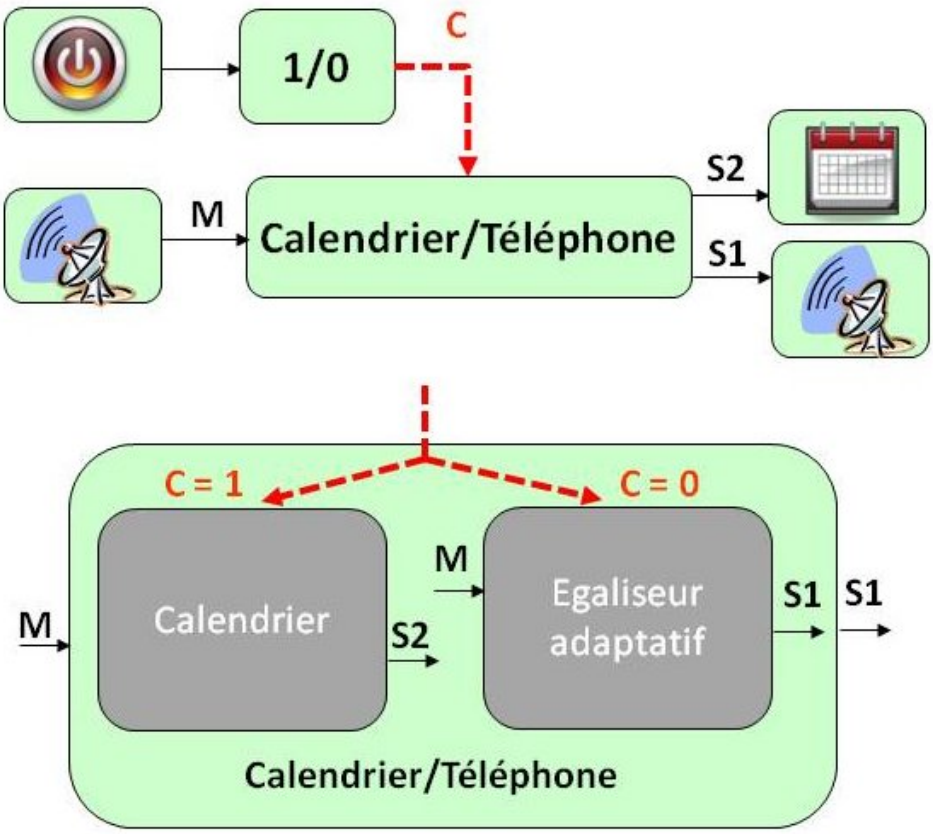
Join : Y(yt, yt-1, yt-2)

Fonction d'adaptation : `adap`

Une entrée vecteur de trois éléments contenant les coefficients h_t deux entrées scalaires correspondant à l'erreur e et x_t, une sortie scalaire $h_t = h_{t-1} + \mu\, e\, x_t$ μ est un coefficient d'adaptation.

Modèle d'algorithme

Exemple : smartphone simplifié



Langages de spécification fontionnelle

Langages généraux : ASSEMBLEUR, FORTRAN, PASCAL, C, etc.

ASSEMBLEUR
FORTRAN, PASCAL
C, C++, JAVA
SystemC,VHDL
MODULA, SIMULA
LISP, CAML
ADA, LTR, GRAFCET
...

Ces langages sont plus ou moins bien adaptés à la spécification fonctionnelle d'algorithmes permettant d'exprimer du parallélisme potentiel et du temps.

Langages de spécification fontionnelle

Langages synchrones : Esterel, Lustre, Scade, Signal, StateCharts, SyncCharts

Issus des langages CSP (Communicating Sequential Processes), CCS (Calculus of Communicating Systems), TLA (Temporal Logic of Actions), etc., les **langages synchrones** possèdent les notions de **parallélisme potentiel** (concurrence) et de **temps logique**.

De plus ils ont une **sémantique mathématique** fondée sur du **calcul d'horloges logiques** qui permet de faire des **vérifications formelles** sur les programmes en termes de parallélisme potentiel et de temps logique.

Ils ont les caractéristiques suivantes :

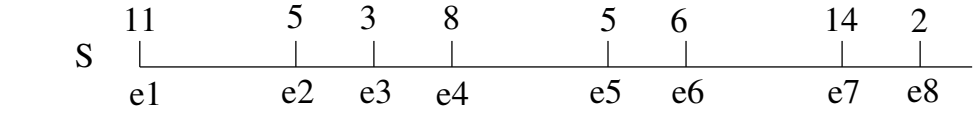
- ESTEREL, STATECHARTS, SYNCCHARTS : impératif, flot de contrôle, calcul d'horloge fonctionnel, horloge maximale donnée (tick) ;
- LUSTRE, SCADE : déclaratif, flot de données, calcul d'horloge fonctionnel, horloge maximale donnée (tick) ;
- **SIGNAL** : déclaratif, flot de données, calcul d'horloge relationnel, horloge maximale calculée par le compilateur.

Langage de spécification fonctionnelle synchrone SIGNAL

Relations entre événements 1/4

SIGNAL est un langage **flot de données synchrone** permettant de spécifier des relations entre des **événements valués**, chacun d'eux étant un élément d'un ensemble infini appelé **signal** et prenant sa valeur dans un ensemble tel que les réels, les entiers, les booléens, etc. Un signal est associé à chaque entrée (resp. sortie) du système de contrôle correspondant à la sortie (resp. entrée) d'une opération capteur (resp. actionneur), ainsi qu'à chaque entrée et sortie des autres opérations du graphe flot de données qui spécife le système décrit en Signal.
Il y a **quatre types** de relation :

1. une **relation de précédence** entre deux événements valués d'un même signal permettant de définir leur ordre d'apparition dans le signal. C'est une relation d'**ordre total strict** que l'on peut représenter par un **diagramme temporel logique** comme ci-dessous ;

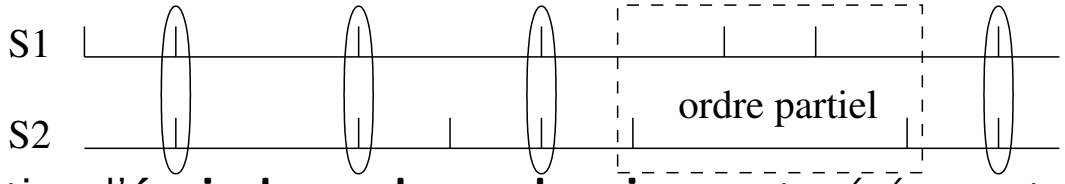


avec e1 < e2 < e3 < e4 < e5 < e6 < e7 < e8 et les ei appartiennent aux entiers

Langage de spécification fonctionnelle synchrone SIGNAL

Relations entre événements 2/4

2. une **relation de synchronisme** entre deux événements de deux signaux différents. Deux événements sont **synchrones** quand ils sont **présents** au **même instant logique**. Si un des événements est présent alors qu'il n'a pas d'événement synchrone sur l'autre signal, l'autre événement est dit **absent**. C'est une relation d'équivalence (symétrique, transitive, réflexive). Tous les événements synchrones appartiennent à une classe d'équivalence appelée **instant logique** t.

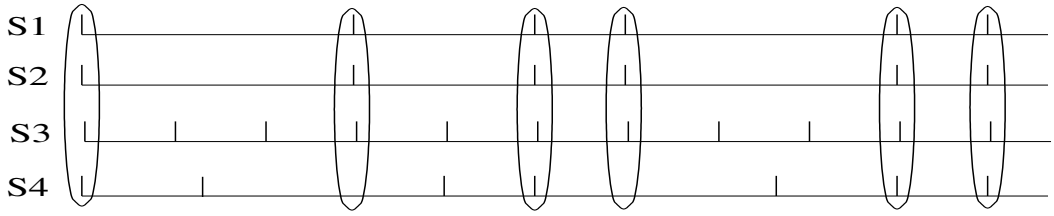


La relation d'**équivalence de synchronisme** entre événements de signaux différents combinée avec la relation d'**ordre total strict de précédence** entre événements d'un même signal, ne permet pas **toujours** de définir un ordre total sur les événements de signaux différents, mais seulement **un ordre partiel** car **parfois** on ne sait pas si l'événement d'un signal précède ou succède celui d'un autre signal ;

Langage de spécification fonctionnelle synchrone SIGNAL

Relations entre événements 3/4

3. une **relation de synchronisme** entre deux signaux quand tous leurs événements sont deux à deux synchrones, c.a.d. présents aux mêmes instants logiques deux à deux (extension de la relation précédente). C'est aussi une relation d'équivalence. Tous les signaux **synchrones** appartiennent à une classe d'équivalence appelée **horloge logique** ;

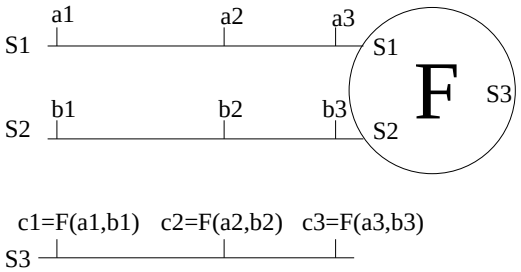


$S1$ et $S2$ ont la même horloge car ils sont synchrones. **Tous** les événements de $S1$ et $S2$ sont synchrones avec certains événements de $S3$ qui a plus d'événements, donc l'horloge de $S1$ et $S2$ est incluse dans l'horloge de $S3$ notée **plus grande** $>$. En revanche, tous les événements de $S3$ ne sont pas synchrones avec ceux de $S4$ et tous les événements de $S4$ ne sont pas synchrones avec ceux de $S3$, donc l'horloge de $S4$ n'est pas plus grande ou plus petite que celle de $S1$ et $S2$ et celle de $S3$. Les horloges de $S1$, $S2$, $S3$ et $S4$ ne sont pas comparables. Donc $\geq$ est une **relation d'ordre partiel** sur les horloges qui n'est pas strict car deux horloges peuvent être égales ;

Langage de spécification fonctionnelle synchrone SIGNAL

Relations entre événements 4/4

4. **4 relations d'entrée-sortie** définissant les **4 instructions de base** du langage. Par exemple la **fonction immédiate** entre des signaux d'entrée et de sortie d'une opération, extension aux signaux des fonctions sur valeurs.



Hypothèse des langages synchrones : **lorsqu'une instruction produit un événement celui-ci est synchrone avec le ou les événements, eux aussi synchrones, qu'elle a consommés**. Comme on ne considère pas le matériel dans la spécification fonctionnelle, cela revient à **ne pas considérer la durée** des instructions et des transferts de données, donc des réactions. Ainsi les signaux d'entrée d'une **fonction immédiate** doivent être synchrones entre eux et sont synchrones avec les signaux de sortie.

De plus, une instruction doit être **causale**, c.a.d. que ses sorties ne doivent pas dépendre d'elles mêmes créant, ainsi, un cycle de dépendance de donnée.

Langage de spécification fonctionnelle synchrone SIGNAL

Horloges des signaux

L'horloge associée à un signal X, notée $P(X)$, est un ensemble strictement ordonné de booléens prenant les valeurs `vrai` quand X est **présent** et `faux` quand X est **absent**, ceci **relativement à d'autres signaux**.

Deux horloges peuvent être comparées par la **relation d'ordre partiel** $\geq$.

Ainsi un signal booléen B, valant `vrai` ou `faux`, a une horloge P(B) qui vaut, elle aussi, `vrai` ou `faux`. On appelle $T(B)$ l'horloge du signal booléen B quand B est `vrai` et on a par exemple $P(X) \geq T(X < 0)$.

Les horloges étant des ensembles, on peut définir leur union et leur intersection. En notant $\cap$ par $\ll \ \gg$ et $\cup$ par $\ll + \gg$ on peut écrire :
$P(X) = P(Y)P(Z)$ $\quad$ $P(X) = P(Y) + P(Z)$ $\quad$ $P(X) \geq P(Y)$

Un programme ou **processus SIGNAL** correspond à la composition avec le caractère $\ll \mid \gg$ d'instructions ou **processus élémentaires** et/ou de processus (encapsulation, modularité). Les **identités de noms** entre des noms de signaux de sortie et des noms de signaux d'entrée induisent des dépendances qui définissent un **ordre partiel d'exécution** des instructions.

Langage de spécification fonctionnelle synchrone SIGNAL

Quatre instructions de base appelées « processus élémentaires »

Syntaxe	**Equation horloge**	**Relation E-S et types**
Fonction immédiate (y1,..,yn) := f(x1,..,xm)	P(y1)=... P(yn)= P(x1)=... P(xm)	y=f(x) types indifférents
Retard zx := x $ n zx init k	P(zx)=P(x)	zx(t)=x(t-1) pour n=1 x, zx mêmes types
Sous-échantillonnage y := x when b	P(y)=P(x)T(b)	y=x quand x et b présents et b vrai x type indifférent et b booléen
Mélange prioritaire y := x0 default x1	P(y)=P(x0)+P(x1)	y=x0 quand x0 présent et x1 absent **ou** y=x1 quand x1 présent et x0 absent **ou** y=x0 quand x0 et x1 présents x0, x1 mêmes types

```
(| y1 := e when e<=0  %P(y1)=P(e)T(e<=0)=T(e<=0) car P(e)>=T(e<=0)%
 | y2 := e when e>0   %P(y2)=P(e)T(e>0)=T(e>0) car P(e)>=T(e>0)%
 | y3 := y1 + y2      Peut-on écrire cela ?
 |)   | y3 := y1 + y2      NON ! erreur de compilation car P(y1) ≠ P(y2)
 |)
```

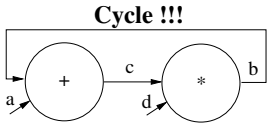
Langage de spécification fonctionnelle synchrone SIGNAL

Graphe flot de données d'un processus

Un processus SIGNAL peut être modélisé par un **graphe flot de données répété conditionné factorisé** dans lequel :

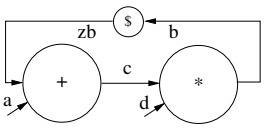
- chaque **sommet**, comportant des **entrées** et des **sorties**, représente une instruction (processus élémentaire) SIGNAL. À chacune de ses sorties est associée une **équation d'horloge** définissant son horloge en fonction des horloges de ses entrées ;
- chaque **arc** ou **dépendance** reliant une entrée à une sortie, représente un signal, une horloge est associée à chaque entrée et chaque sortie ;
- un sommet s'exécute quand les horloges de ses entrées, auxquelles sont associées des équations d'horloge, sont présentes ;
- une suite d'arcs dont le premier sommet et le dernier sommet sont identiques, conduit à un cycle dans le graphe et donc à un programme **non causal** s'il ne comprend pas de **retard**. Ces cycles sont **interdits**.

```
| c := a + b
| b := d * c
```



$\Rightarrow$

```
| zb := b $ 1
| c := a + zb
| b := d * c
```



Langage de spécification fonctionnelle synchrone SIGNAL

Fonctionnement du compilateur

Le compilateur effectue des **vérifications** :

- classiques sur valeurs (type, indices tableaux, division par zéro, etc.),
- **formelles**, garantissant des **propriétés temporelles logiques** :
  - sur les arcs pour s'assurer qu'il n'y pas de **cycles sans retard** ;
  - sur le système d'équations d'horloges pour **montrer** que l'horloge de chaque signal de sortie **est égale** à une expression d'horloges de signaux d'entrée. On dit alors que l'horloge d'un signal est **déterminée**. Si le système d'équations ne peut pas se résoudre, il y a deux cas :
    - trop de contraintes (redondance dans le calcul d'horloge) ;
    - pas assez de contraintes : il faut donner une horloge déterminée aux signaux dont l'horloge est indéterminée avec l'instruction de contrainte d'horloge : `horloge indeterminee ^= horloge determinee`.

Il génère, par exemple, un programme séquentiel en C qui permet de faire une **simulation fonctionnelle et temporelle logique** garantissant que l'**ordre des événements de sortie est cohérent** par rapport à l'ordre des événements d'entrée qui les ont provoqués. Cette simulation **est indépendante** de la durée des instructions.

Langage de spécification fonctionnelle synchrone SIGNAL

Diagrammes temporels logiques des signaux 1/3

Processus $\cdots$ Programme Signal système de contrôle : $\perp$= absent

X	>	1.0
Y	>	$\perp$
Z	>	2.5

Processus élémentaires monochrones : signaux avec mêmes horloges

Fonctions immédiates : extensions aux signaux des fonctions sur valeurs, fonctions arithmétiques usuelles, fonctions booléennes, `NOT, AND, OR` et comparaisons `=, /=, <, <=, >, >=`

`X := A + B`

A :	1	1	3	4
B :	2	1	6	3
X :	3	2	9	7

`B3 := B1 AND B2`

B1 :	T	T	F	F
B2 :	F	T	F	T
B3 :	F	T	F	F

`B := A >= 0`

A :	3	0	-2	-6
B :	T	T	F	F

Langage de spécification fonctionnelle synchrone SIGNAL

Diagrammes temporels logiques des signaux 2/3

`ZX := X $ 1` Retard d'un instant logique de `Z` par rapport à `X`.

X :	2	1	5	4	3
ZX :	0	2	1	5	4

ZX initialisé à 0

`^X` Horloge de `X`, signal type `event vrai=present faux=absent`.
Le type `event` est utile lorsqu'on s'intéresse **seulement à l'horloge** d'un signal sans s'intéresser aux valeurs des événements qui le constituent.

X :	1	2	3	4	5
^X :	T	T	T	T	T

`X ^= Y` Contrainte d'horloge, `X` et `Y` de type `event`. Ce n'est pas une instruction mais une directive de compilation. Il lui est associée l'équation d'horloge `P(X)=P(Y)`. Lors de la compilation dans le calcul d'horloges, l'horloge indéterminée de `X` prend l'horloge déterminée de `Y` qui peut être une expression sur signaux.

Langage de spécification fonctionnelle synchrone SIGNAL

Diagrammes temporels logiques des signaux 3/3

Processus élémentaires polychrones : horloges différentes

`X := A when B` Sous-échantillonnage de `A` par `B vrai`

A :	1	2	3	4	⊥	5	6	⊥	7	8
B :	F	T	⊥	T	T	F	T	F	⊥	T
X :	⊥	2	⊥	4	⊥	⊥	6	⊥	⊥	8

La sortie `X` prend la valeur de l'entrée `A` quand celle-ci est présente et que l'entrée `B` est présente **et** `B vrai`.

`Y := X0 default X1` Mélange de `X0` et `X1` avec `X0` prioritaire

X0 :	1	3	⊥	5	6	⊥	8	9
X1 :	⊥	2	4	2	⊥	7	9	6
Y :	1	3	4	5	6	7	8	9

La sortie `Y` prend la valeur de l'entrée `X0` quand celle-ci est présente **ou** de l'entrée `X1` quand celle-ci est présente **ou** `X0` quand les deux sont présentes.

Langage de spécification fonctionnelle synchrone SIGNAL

Signaux constants

L'horloge d'un signal constant est déterminée par son contexte.

Pas de problèmes avec les fonctions immédiates et le `when` :
`X := A + 1 => P(X)=P(A)=P(1)` : `X` à l'horloge de `A`
`X := 1 when B => P(X)=P(1)T(B)` : `X` vaut 1 à l'horloge de `B vraie`

Parce que le signal de gauche d'un `default` est prioritaire **ATTENTION** :
`Y := X0 default 1 => P(Y)=P(X0)=P(1)` : `Y=X0` à l'horloge de `X0`
`Y := 1 default X1 => P(Y)=P(X1)=P(1)` : `Y=1` à l'horloge de `X1`

Langage de spécification fonctionnelle synchrone SIGNAL

Règles de simplification utilisées par le compilateur

L'ensemble des horloges d'un programme SIGNAL, muni de la relation d'**ordre partiel** $\geq$ et des lois internes d'**union** (borne supérieure des horloges), notée additivement, et d'**intersection** (borne inférieure des horloges), notée multiplicativement, forment une **structure algébrique de treillis** avec les propriétés suivantes :

```
Priorité à l'intersection : P(X)P(Y)+P(Z)=(P(X)P(Y))+P(Z)
Commutativité : P(X)+P(Y) = P(Y)+P(X); P(X)P(Y) = P(Y)P(X)
Distributivité : P(X)+(P(Y)P(Z))=(P(X)+P(Y))(P(X)+P(Z))
Distributivité : P(X)(P(Y)+P(Z))=(P(X)P(Y))+(P(X)P(Z))
Idempotence : P(X)+P(X) = P(X); P(X)P(X) = P(X)
Absorption : P(X)(P(X)+P(Y)) = P(X); P(X)+(P(X)P(Y)) = P(X)
```

On en déduit les propriétés suivantes qui seront **très utiles** dans la suite :

```
P(X)+P(Y) = P(X) <=> P(X) >= P(Y) inégalité : P(X) est indéterminée
P(X)P(Y)  = P(X) <=> P(X) <= P(Y) inégalité : P(X) est indéterminée
P(X)+P(Y) >= P(X) et P(X)+P(Y) >= P(Y)
P(X)P(Y) <= P(X) et P(X)P(Y) <= P(Y)
B boolean P(B) >= T(B) et par exemple P(X) >= T(X=0)
```

Langage de spécification fonctionnelle synchrone SIGNAL

Syntaxe des processus ou programmes 1/3

Les éléments terminaux du langage (mots-clés) sont soulignés, les éléments de syntaxe entre crochets $\ll$ [...] $\gg$ sont optionnels, $\ll$ / $\gg$ indique une alternative.

processus = process nom = [{ paramètres }]
(? signaux-entrée ! signaux-sortie)
corps
[where signaux-locaux
[processus ; processus ...]
] end

processus = function nom = (? signaux-entrée ! signaux-sortie)

Langage de spécification fonctionnelle synchrone SIGNAL

Syntaxe des processus ou programmes 2/3

paramètres = type nom , nom . . . ; type nom , nom . . .
signaux-entrée = signaux-sortie = type nom , . . . ; type nom , . . .
signaux-locaux = type nom [init val / expr-tableau] , nom . . . ; . . .
type = [[val , . . .]] type-scalaire
type-scalaire = event / boolean / integer / real / dreal
corps = (| inst | inst | . . . |)
inst = nom := expr
inst = (nom , . . .) := appel-sous-processus
inst = nom := expr-tableau
inst = tableau-processus
expr = **expression combinant processus élémentaires et appel de sous-processus**
appel-sous-processus = nom [{ val , . . .}] (expr , . . .)
nom = **suite de caractères alpha-numériques**
val = **expression ne contenant que des constantes et/ou des paramètres**

Langage de spécification fonctionnelle synchrone SIGNAL

Syntaxe des processus ou programmes 3/3

% les commentaires sont mis entre pourcent %

Un **expression** combine plusieurs processus ou processus élémentaires. On peut ne pas pas utiliser de parenthèses sachant que la fonction immédiate est plus prioritaire que le `when`, lui même plus prioritaire que le `default`. Par exemple l'expression `y := (a when (b > 0) ) default (z + 1)` peut s'écrire `y := a when b > 0 default z + 1`

L'instruction de composition ≪ | ≫ n'induit qu'un ordre partiel sur l'exécution des processus. Les dépendances sont déduites des noms des signaux. Un signal de sortie est connecté par un arc au signal d'entrée de même nom, ou par plusieurs arcs aux signaux d'entrée de mêmes noms (**diffusion de donnée autorisée**). Plusieurs signaux de sortie **ne peuvent pas** être connectés à un même signal d'entrée (**confusion de données interdite**).

Langage de spécification fonctionnelle synchrone SIGNAL

Exemple représentatif de la syntaxe

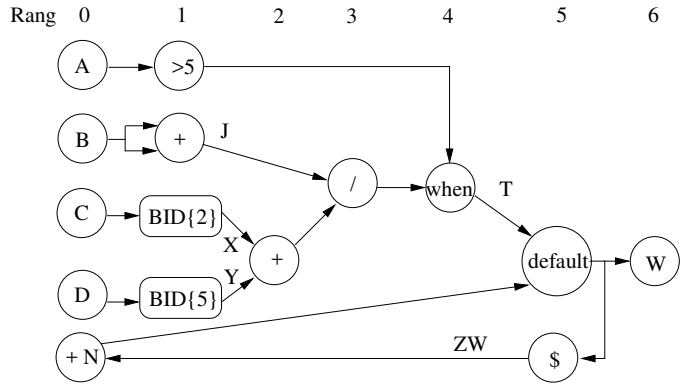
```
process EXEMPLE = {integer N}  %nom et paramètres du processus%
( ? integer A,B,C,D ! real W ) %déclaration signaux entrée/sortie%
(| X := BID{2}(C)              %appel BID paramètre local 2, P(X)=P(C)%
 | J := B+B                    %P(J)=P(B)%
 | Y := BID{5}(D)              %appel BID paramètre local 5, P(Y)=P(D)%
 | T := (X+Y)/J when A>5       %P(C)=P(D)=P(B) et P(T)=P(B)T(A>5)%
 | ZW := W $ 1                 %P(ZW)=P(W)%
 | W := T default ZW+N         %P(W)=P(B)T(A>5)+P(W)=>P(W)>=P(B)T(A>5)(1)%
 | ^W ^= ^B when A>5|)         %P(W)=P(B)T(A>5) vérifie (1)%
where                          %déclaration signaux locaux et sous-processus%
 integer J, X, Y, ZW init 0 ; real T ;
 process BID = {integer PARAM} %nom et paramètres du sous-processus%
  (? integer X ! integer Y )   %déclaration signaux entrée/sortie locaux%
  (| Y := X*PARAM |)           %P(Y)=P(X)%
end
```

Un paramètre est un **signal d'entrée constant** déclaré et utilisé dans un **processus** (`{integer N}`, `W := T default ZW+N`) dont la valeur est donnée dans un fichiers de paramètres, ou déclaré et utilisé dans un **sous-processus** (`{integer PARAM}`, `Y := X*PARAM`) dont la valeur est donnée à l'appel des sous-processus (`BID{2}` et `BID{5}`).

Langage de spécification fonctionnelle synchrone SIGNAL

Graphe flot de données du processus EXEMPLE

Programme Signal $\Leftrightarrow$ Graphe flot de données.



Les sommets sans prédécesseurs A, B, C, D (resp. sans successeurs W) représentent des capteurs (resp. des actionneurs) connectés à d'autres sommets par des signaux de mêmes noms que les sommets capteurs (resp. des actionneurs). **On ne tient pas compte des contraintes d'horloges** utiles uniquement pour le compilateur et qui ne seront donc pas implantées. **On ne tient pas compte du retard** car son coût est négligé bien qu'il soit implanté.

Plus grand stable $\{A, B, C, D, +N\}$ et $\{> 5, +, BID2, BID5, +N\}$ de $Card = 5$.

Ici le nombre de processeurs **potentiellement en parallèle** est donc de 5.

Langage de spécification fonctionnelle synchrone SIGNAL

Exemples de processus 1/4

```
% Compteur des événements présents de top %

process cpt = {}
(? event top ! integer n)

(| zn := n $ 1              % P(zn)=P(n) %

 | n := zn + 1              % P(n)=P(zn) %

 | ^n ^= top                % P(n)=P(top) %
 |)
where integer zn init 0
end
```

Ecrire `| n := zn + 1 when top |` $\Rightarrow$ `P(n)=P(zn)T(top)` $\Rightarrow$ `T(top)>=P(n)` ce qui ne permet pas de déterminer l'horloge de `n`.

Langage de spécification fonctionnelle synchrone SIGNAL

Exemples de processus 2/4

```
% Mémorisation d'un signal e à l'aide d'un autre signal m %
% s prend la valeur de e quand e est present et prend %
% la valeur precedente de s quand e est absent et m present %

process mem = {}
(? integer e; event m ! integer s)
(| zs := s $ 1              % P(zs)=P(s) %

 | s := e default zs        % P(s)=P(e)+P(zs) => P(s)>=P(e) (1) %

 | ^s ^= ^e default m       % P(s)=P(e)+P(m) verifie (1) %
 |)
where integer zs init 0
end
```

Comme précédemment écrire `| s := e default zs when m |` ne permet pas de déterminer l'horloge de `s`.

Langage de spécification fonctionnelle synchrone SIGNAL

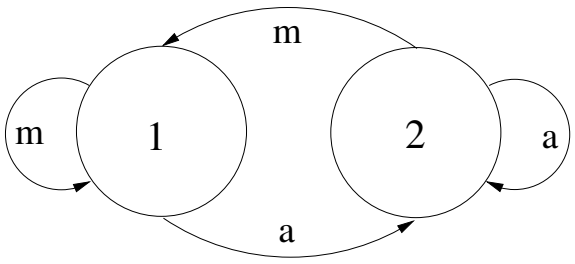
Exemples de processus 3/4

```
% Compteur des événements présents de top %
% remis à zéro quand raz présent et vrai %

process cptRaz = {}
(? event top; boolean raz ! integer n)
(| zn := n $ 1                          % P(zn)=P(n) %

 | n := 0 when raz default zn +1        % P(n)=T(raz)+P(zn)
                                          => P(n)>=T(raz) (1) %

 | ^n ^= when ^raz default top          % P(n)=T(raz)+P(top)
 |)                                       verifie (1) %
where integer zn init 0
end
```

Écrire `| n := 0 when raz default zn +1 when top |` ne permet pas de déterminer l'horloge de `n`.

Langage de spécification fonctionnelle synchrone SIGNAL

Exemples de processus 4/4




```
% Automate 2 états : marche (1) et arrêt (2), deux entrées %

process automate = {}
(? event m, a ! integer x)
 | x := 2 when a default 1 when m   % P(x)=T(a)+T(m) %
 |)
```

Spécification des Architectures

Généralités

Architecture distribuée, parallèle, multiprocesseur, multi(pluri)cœur

Ces architectures sont formées de plusieurs processeurs (cœurs), connectés par des média de communication **point-à-point** ou **multi-point** (bus), qui communiquent par **des mémoires distribuées** en faisant du **passage de messages**, ou par **une mémoire partagée**. On a quatre types d'architecture :

- **distribuée** : les communications se font par passage de message, les processeurs sont de types différents (GRID) ;
- **parallèle** : les communications se font par mémoire partagée ou par passage de message, les processeurs sont de mêmes types ;
- **multiprocesseur** : les communications se font par mémoire partagée, les processeurs sont de mêmes types ou de types différents ;
- **multi(pluri)cœur** : les processeurs sont sur un même circuit intégré (puce) et communiquent par mémoire partagée (multi) et/ou par un réseau (pluri).

La façon dont les processeurs et média sont connectés conduit à différentes **topologies** : anneau, étoile, matrice, hypercube, complètement connecté, etc.

Une **route** correspond à une suite de couples commençant par un processeur suivi d'un médium ou d'un routeur dans le cas d'un réseau, le couple processeur médium étant répété un certain nombre de fois, et se terminant par un processeur.

Généralités

Parallélisme, architecture multicomposant

Types de parallélisme (classification de Flynn 1996) :

- **de contrôle** : (MIMD) les processeurs exécutent en parallèle des opérations différentes sur des données qui peuvent être différentes ;
- **de données** : (SIMD) les processeurs exécutent en parallèle la même opération sur des données différentes ;
- **de flot ou flux** : (MISD) travail à la chaîne **répétitif**, pipeline, les processeurs exécutent des opérations différentes sur la même donnée.

Une **architecture multicomposant** est hétérogène, elle peut offrir des types de parallélismes différents et peut être formée de plusieurs composants différents de chaque type suivant :

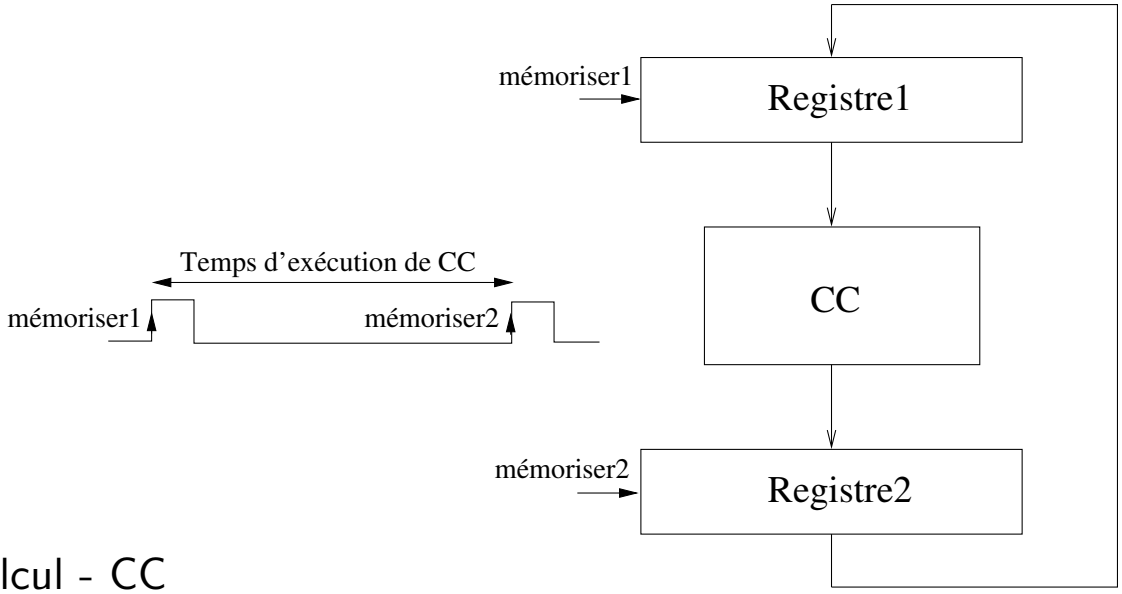
- **processeur** : composant programmable qui exécute séquentiellement les opérations (séquence d'instructions) d'un programme ;
- **circuit intégré spécifique** (ASIC, FPGA) : composant non programmable qui exécute **une seule** opération ;
- **médium de communication** : point-à-point ou multi-point (bus).

Modèle d'architecture distribuée multicomposant

RTL

Modèle RTL : Register Transfert Level

Transfert de données entre registres à travers un circuit combinatoire (CC).



- Calcul - CC
- Mémoire (cache, interne, externe) - Registre
- Moyen de communication (bus, lien) - Chemin de données ($\longrightarrow$)

Modèle d'architecture distribuée multicomposant

Machine séquentielle

Une architecture multicomposant est fondée sur la notion de **machine séquentielle** qui est un automate fini avec sorties.

- **Automate fini** (accepteur ou reconnaisseur) : machine à états finie (FSM) (E, X, i, f, t)

  E ensemble fini de symboles d'entrée (événements d'entrée)
  X ensemble fini d'états, états initiaux $i \subset X$, états finaux $f \subset X$
  X est associé à une mémoire qui contient le passé de l'automate
  t fonction de transition $t : E \times X \to X \qquad x_{t+1} = t(e, x_t)$

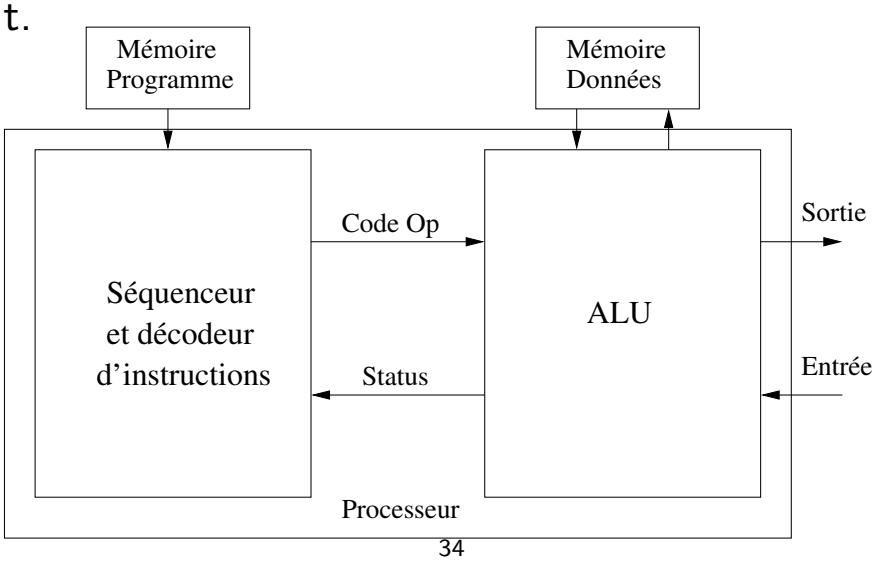
- **Automate fini avec sorties** (transducteur) : machine séquentielle dans le domaine de l'électronique numérique (E, X, i, f, t, S, s)

  S ensemble fini de symboles de sortie (événements de sortie)
  t fonction de transition, s fonction de sortie exécute le CC
  Mealy $s : E \times X \to S \quad w = s(e, x_t)$ Moore $s : X \to S \quad w = s(x_t)$

Modèle d'architecture distribuée multicomposant

Processeur

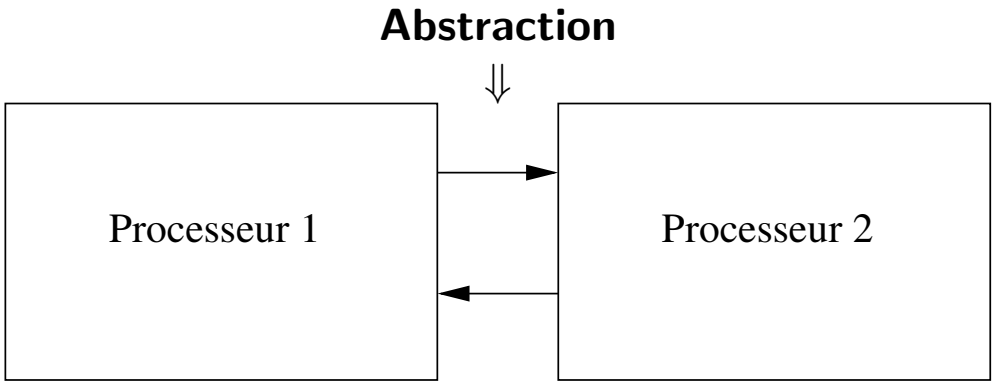
Un processeur peut être modélisé par deux machines séquentielles connectées : un séquenceur et une ALU.
Le **séquenceur** lit l'état produit par l'ALU, lit une instruction dans la mémoire programme, la décode et écrit son code opératoire dans l'ALU.
L'**ALU** lit le code opératoire et exécute l'opération correspondante qui lit ses opérandes dans la mémoire de donnée (resp. entrée) et y (resp. sortie) écrit le résultat.



Modèle d'architecture distribuée multicomposant

Modèle simple d'architecture distribuée ou parallèle

Deux processeurs, chacun d'eux étant modélisé par deux machines séquentielles connectées, connectés par des liens point-à-point forment une **architecture distribuée ou parallèle**. Pour simplifier l'architecture on peut faire **abstraction** de l'intérieur de chaque processeur.



Une architecture multicomposant est spécifiée à l'aide d'un **modèle de graphe orienté** dont les sommets et les arêtes sont les suivants :

Modèle d'architecture distribuée multicomposant

Modèle de graphe orienté multicomposant AAA 1/2

- **sommet** : machine séquentielle atomique de **quatre types** :
  1. **opérateur** séquence des opérations (ALU, FPU...) et des dépendances quand il n'y a pas de communicateurs ;
  2. **communicateur** séquence des dépendances (DMA...) ;
  3. **mémoire** :
     - à accès aléatoire (RAM) :
       - de données (D) ou de programme (P) ;
       - de communication de **données partagées**, accédée par plusieurs communicateurs (**arbitrage**) pour faire de la mémoire partagée ;
     - à accès séquentiel (SAM) : de communication de **données distribuées**, accédée par un seul communicateur pour faire du passage de messages, point-à-point ou multi-point (bus), avec ou sans diffusion ;
  4. **mux, demux** :
     - **mux** : accès de plusieurs opérateurs et/ou communicateurs à une mémoire partagée => **arbitrage** ;
     - **demux** : accès d'un opérateur ou communicateur à plusieurs mémoires => **routage** ;
- **arête** : regroupe deux arcs de sens opposés, connexion bidirectionnelle entre **entrées/sorties de sommets**. **Une arête ne peut pas connecter des sommets de même type, sauf pour les sommets de type mux/demux.**

Modèle d'architecture distribuée multicomposant

Modèle de graphe orienté multicomposant AAA 2/2 et abstraction 1/2

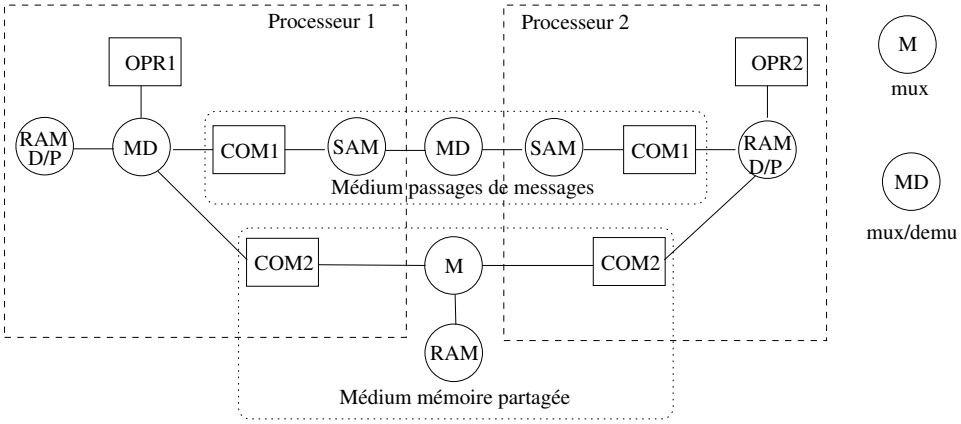
Chaque composant d'une architecture AAA est un sous-graphe formé, pour un :
- **processeur** : un opérateur exécutant plusieurs opérations, une **RAM P** et une RAM D, et plusieurs mux/demux, SAM, communicateurs ;
- **circuit intégré spécifique** : un opérateur exécutant **une seule** opération, une RAM D, et plusieurs mux/demux, SAM, communicateurs ;
- **médium passage messages** : un mux/demux et plusieurs communicateurs, SAM ;
- **médium mémoire partagée** : un mux et plusieurs communicateurs et une RAM ;
- **bus** : un mux/demux (connexion de plusieurs processeurs avec un bus) ;
- **routeur** : un demux (connexion d'un processeur avec un routeur N,S,E,O).

Abstraction

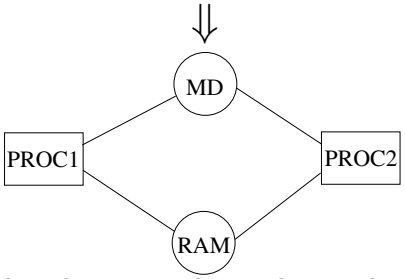
Un sous-graphe représentant un processeur ou un circuit intégré spécifique est **abstrait** en **un seul sommet** opérateur. Les mémoires programme et données, et les mux/demux sont cachés. Un sous-graphe représentant un médium est **abstrait** en **un seul sommet** médium. Les communicateurs et les SAM sont cachés. Pour les média par passage de messages **ce sommet est un mux/demux** qui se réduit à une connexion bidirectionnelle pour une liaison point-à-point qui ne nécessite ni arbitrage ni routage. Chaque communicateur et sa SAM distribuée sont représentés par une **entrée/sortie de l'opérateur**. Pour les média par mémoires partagées **ce sommet est une RAM incluant un mux**. Chaque communicateur est représenté par une **entrée/sortie de l'opérateur**.

Exemple de modèle d'architecture multicomposant

Abstraction 2/2



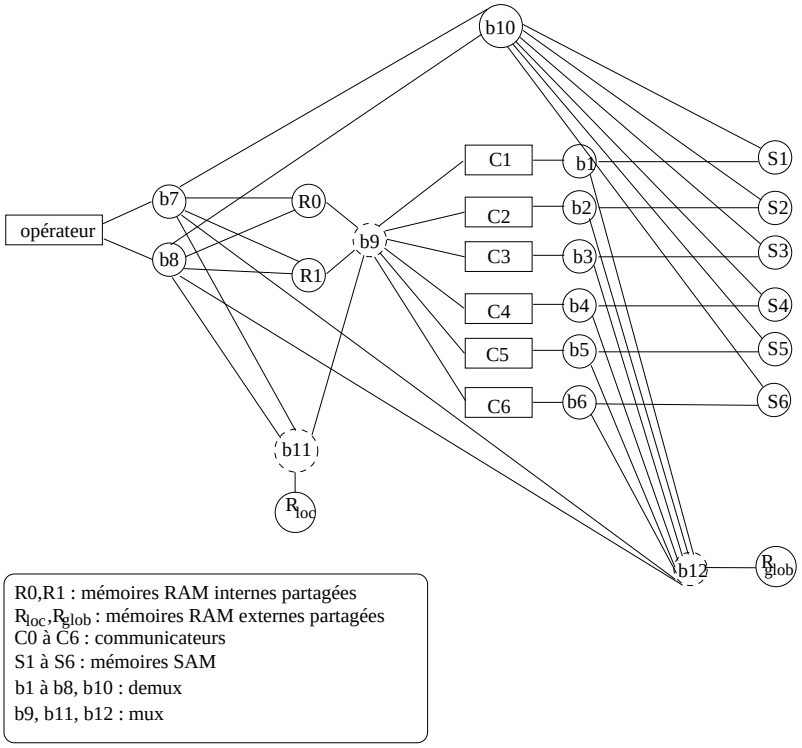
Abstraction



L'abstraction **réduit** la taille du graphe, donc le **temps d'exécution des optimisations** qui peut être très long et leur **précision**.

Exemple de modèle d'architecture multicomposant

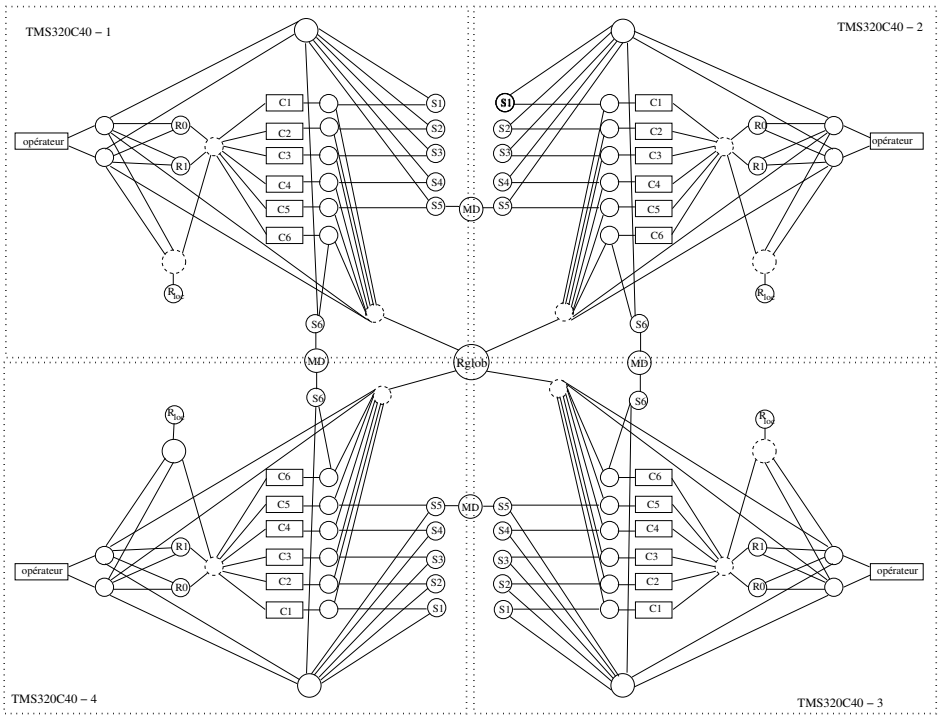
TMS320C40



29 sommets dans le graphe d'architecture

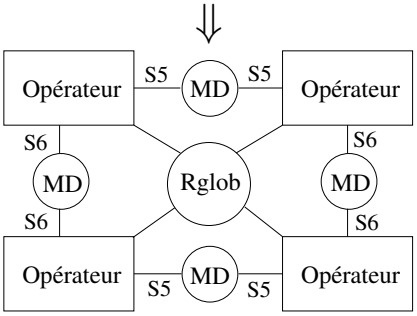
Exemple de modèle d'architecture multicomposant

Quatre TMS320C40 connectés en point-à-point et multi-point



109 sommets

Abstraction



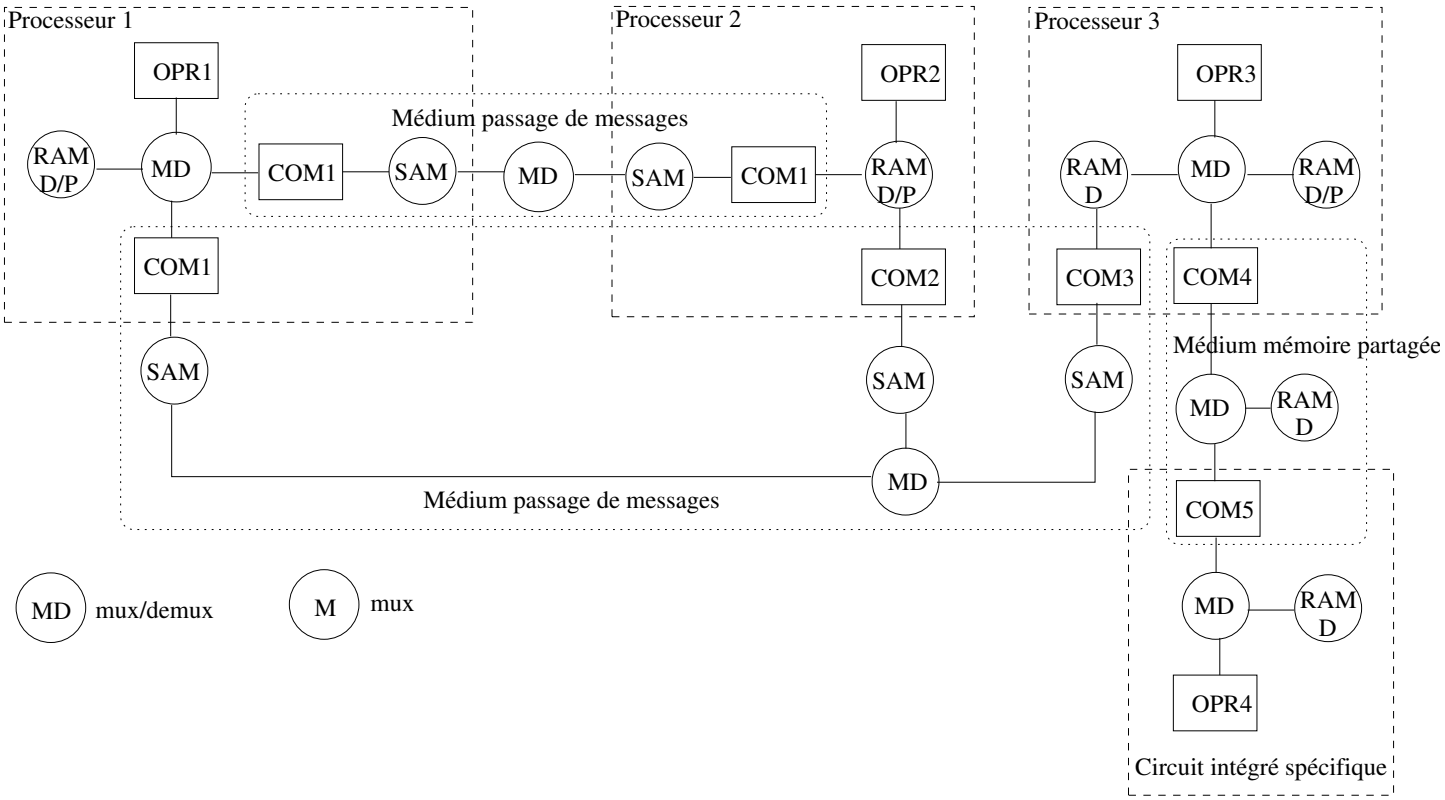
Les mémoires (D/P) et les mux/demux sont cachés, les communicateurs et les SAM sont cachés.

Les communicateurs et les SAM et les communicateurs seulement, sont représentés par les entrées/sorties.

9 sommets

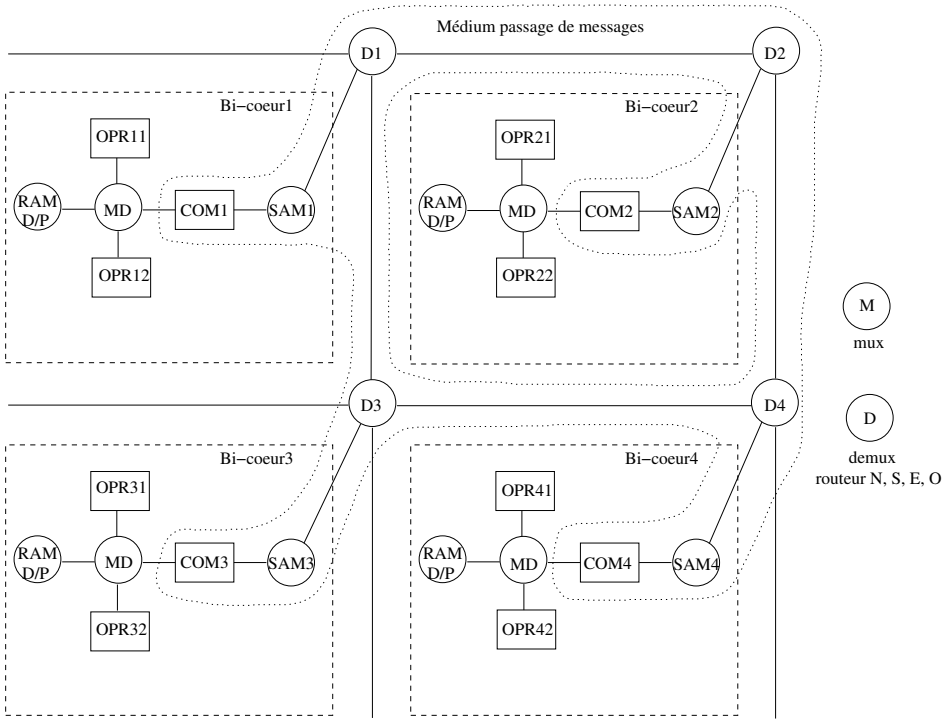
Exemple de modèle d'architecture multicomposant

Trois processeurs et un CIS connectés par passage de messages et mémoire partagée



Exemple de modèle d'architecture multicomposant

Pluricœur : les multicœurs à mémoire partagée sont connectés par un réseau de routeurs



Médium passage de messages $\{(COM1, SAM1), (SAM1, D1), (COM2, SAM2), (SAM2, D2), (COM3, SAM3), (SAM3, D3), (COM4, SAM4), (SAM4, D4), (D1, D2), (D1, D3), (D2, D4), (D3, D4)\}$

Implantation optimisée : Adéquation

Généralités

Distribution et ordonnancement 1/2

L'**implantation** s'effectue à partir des spécifications fonctionnelle et non fonctionnelle (architecture matérielle, caractéristiques temporelles indépendantes et dépendantes de l'architecture). Elle consiste à réaliser une **distribution** et un **ordonnancement** de l'algorithme sur l'architecture.

La **distribution** consiste à distribuer les opérations sur les opérateurs (ressources de calcul) et les dépendances sur les média (ressources de communication). On appelle aussi la distribution : allocation, répartition, partitionnement, placement et migration.

L'**ordonnancement** consiste, pour chaque opérateur, à déterminer l'ordre dans lequel seront exécutées les opérations qui y ont été distribuées, et pour chaque médium l'ordre dans lequel seront exécutées les dépendances qui y ont été distribuées.

L'ordonnancement **doit être compatible** avec l'**ordre partiel induit par les dépendances** du graphe d'algorithme, c.a.d. qu'il ne peut que diminuer le parallélisme potentiel. Il **doit** aussi assurer que les **contraintes temps réel** sur les opérations sont respectées.

Généralités

Distribution et ordonnancement 2/2

Distribution et ordonnancement **sont réalisés par un exécutif** soit **en ligne** qui prend des décisions pendant l'exécution de l'application, soit **hors ligne** qui applique des décisions prises avant l'exécution de l'application.

Les approches **en ligne** permettent de prendre en compte des opérations dont les caractéristiques temporelles ne sont pas totalement connues lors de la spécification, mais elles ont **surcoût important qui est variable** et, donc, ne sont **pas déterministes**.

Les approches **hors** ligne demandent une connaissance détaillée de l'algorithme, de l'architecture et des caractéristiques temporelles, mais elles ont un **surcoût peu important qui est fixe** et sont donc **déterministes**, ce qui les rend bien adaptées au **temps réel critique**.

On privilégiera donc par la suite les **approches hors ligne**, dans lesquelles les résultats de la distribution et de l'ordonnancement peuvent être utilisés pour générer des **exécutifs synthétisés sur mesure**, équivalent à de la compilation, pouvant faire éventuellement appel à un **exécutif résident**.

Généralités

Tâches, ordonnanceur, coût de l'exécutif

Les opérations de l'algorithme lorsqu'elles sont caractérisées temporellement sont appelées « **tâches** » par la communauté temps réel, éventuellement « **tâches dépendantes** » si les opérations ont des dépendances. Elles sont ordonnancées par **l'exécutif, principalement formé d'un ordonnanceur**, qui peut être :

- **en ligne** : déclenché par des **événements** produisant des interruptions ;
- **hors ligne** : déclenché par le **temps** avec un compteur de temps discret ;
- **préemptif** : une tâche peut être interrompue par l'ordonnanceur pour exécuter une tâche de plus haute priorité, puis reprendre son exécution ;
- **non préemptif** : une tâche qui a débuté son exécution doit se terminer pour que l'ordonnanceur puisse exécuter une autre tâche.

L'ordonnanceur est appelé (déclenché) **à chaque activation** et **à chaque terminaison** d'une tâche pour choisir la prochaine tâche à exécuter parmi les tâches prêtes à s'exécuter. **L'exécutif a un coût** égal au temps qu'il faut :

- pour **choisir la prochaine tâche à exécuter** :
  - en ligne : **variable**, car il faut comparer les priorités des tâches prêtes ;
  - hors ligne : **constant**, en lisant dans une **table d'ordonnancement** ;
- pour **traiter les préemptions** pouvant causer **d'autres préemptions** :
  - en ligne : sauvegarde et restauration de contexte ;
  - hors ligne : sauvegarde et restauration de contexte.

Ordonnancement temps réel monoprocesseur

En ligne préemptif 1/10 : caractéristiques d'une tâche

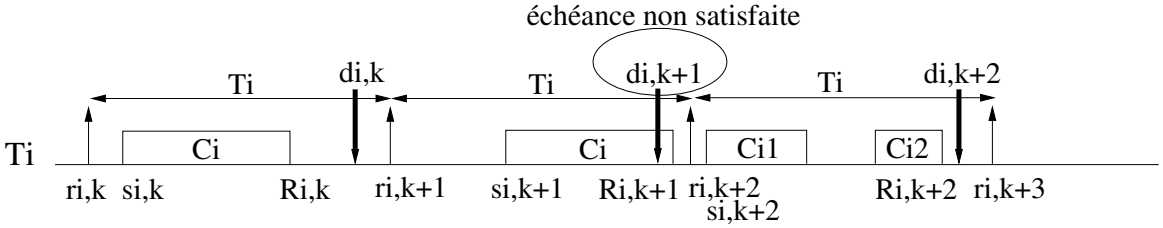
Le modèle classique (Liu et Layland 1973) est fondé sur un **ordonnanceur en ligne** qui gère un ensemble de tâches **indépendantes** où chaque tâche t_i est la répétition infinie d'une « instance » (« travail » ou « job ») t_i^k, avec $k = 1..\infty$.
Une **tâche** t_i est **caractérisée temporellement** par les informations suivantes :

- **date d'activation** ou de réveil r_i^k (release time), si elle est périodique (resp. sporadique) de période (resp. période minimale) T_i on a $r_i^k = r_i^1 + kT_i$;
- **première date d'activation** r_i^1 ou déphasage (offset) : tâche concrète ;
- **date de début d'exécution** (start time) $s_i^k \neq r_i^k$ car coût de l'exécutif ;
- **durée d'exécution pire cas** C_i (WCET Worst Case Execution Time) à laquelle on ajoute une **approximation du coût de l'exécutif** ;
- **échéance relative, délai critique** (deadline) D_i : durée, depuis r_i^k, avant laquelle la tâche i **doit être terminée**, si $D_i = T_i$ **échéance sur requête** ;
- **échéance absolue** à partir de l'origine du temps $d_i^k = r_i^k + D_i$;
- **temps de réponse** R_i^k : durée, depuis r_i^k, où la tâche i **se termine** ;
- **laxité** $l_i^k(t) = d_i^k - (t + C_i(t))$: différence entre l'échéance absolue et la fin de l'exécution de la tâche, $C_i(t)$ durée restant à exécuter.

L'**activation** d'une tâche est en général héritée de la **cadence** d'un capteur et les **temps de réponse** des tâches induisent une **latence** (max R_i sans dépendances, ou $\sum R_i$ avec).

Ordonnancement temps réel monoprocesseur

En ligne préemptif 2/10 : algorithme d'ordonnancement d'un ordonnanceur



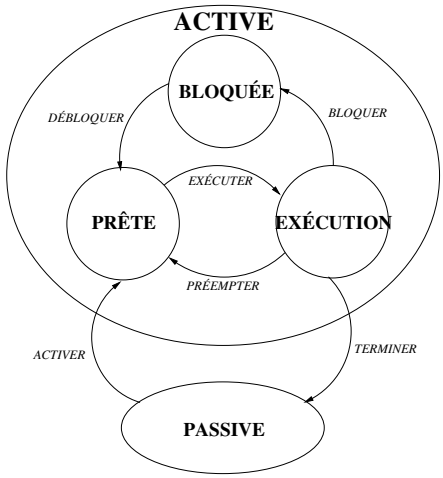
Modèle de tâche de Liu et Layland

L'**ordonnanceur** utilise un **algorithme d'ordonnancement** préemptif ou non préemptif, fondé sur la notion de **priorités** associée à chaque tâche.
La **préemption** augmente en général le **taux d'ordonnançabilité** (nombre d'ordonnancements possibles), mais il faut **prendre en compte son coût** dans le coût de l'exécutif afin que les conditions d'ordonnançabilité soient **effectivement respectées**. Comme certaines préemptions peuvent causer d'autres préemptions, le coût des préemptions peut varier d'une instance à l'autre.
Les **priorités fixes** utilisées par l'ordonnanceur pour déterminer la prochaine tâche à ordonnancer, ne changent pas au cours de l'exécution, elles sont lues dans une table de priorités. Les **priorités dynamiques** sont calculées lors de l'exécution.
Dans les **deux cas** le choix de la prochaine tâche à ordonnancer a un **coût variable**, mais les priorités dynamiques conduisent à un coût de l'ordonnanceur en ligne **plus élevé** car il doit les calculer plutôt que les lire dans une table.

Ordonnancement temps réel monoprocesseur

En ligne préemptif 3/10 : automate d'un ordonnanceur

Un ordonnanceur en ligne préemptif est constitué d'un automate par tâche et d'un gestionnaire des automates de tâches



L'automate d'une tâche a **quatre états** :
- **PRÊTE** à être exécutée car elle a été activée ;
- **EXÉCUTION** en cours d'exécution ;
- **BLOQUÉE** en attente d'une ressource ;
- **PASSIVE** en attente d'une *activation*.

L'automate d'une tâche a **six événements d'entrée** :
- *ACTIVER* : produit par une interruption externe associée à la tâche ;
- *TERMINER* : produit par la tâche elle-même à la fin de son exécution ;
- *EXÉCUTER*, *PRÉEMPTER* : produits par le **gestionnaire** ;
- *BLOQUER*, *DÉBLOQUER* : produits par le **gestionnaire**.

Ordonnancement temps réel monoprocesseur

En ligne préemptif 4/10 : automate d'un ordonnanceur

Ordonnanceur en ligne préemptif à priorités fixes

Un seul des automates de tâches est dans l'état **EXÉCUTION**.

L'ordonnanceur est appelé à chaque activation (événement *ACTIVER*) et terminaison d'une tâche (événement *TERMINER*).

Sur l'événement *ACTIVER* d'une tâche, cette tâche passe de l'état **PASSIVE** à **PRÊTE**, le **gestionnaire** ou un dispositif matériel externe au processeur **compare** sa priorité avec la priorité de la tâche en cours, seule tâche à être dans l'état **EXÉCUTION**.

Si la priorité de la tâche qui vient de s'activer est supérieure à celle de la tâche en cours, le **gestionnaire** sauvegarde le contexte de la tâche en cours, il produit l'événement *PRÉEMPTER* pour son automate qui passe alors de l'état **EXÉCUTION** à **PRÊTE**, il produit un événement *EXÉCUTER* pour la tâche qui vient de s'activer qui passe de l'état **PRÊTE** à **EXÉCUTION** et enfin il lance son exécution.

Ordonnancement temps réel monoprocesseur

En ligne préemptif 5/10 : automate d'un ordonnanceur

Sur l'événement *TERMINER*, produit par la tâche en cours, celle-ci passe de l'état **EXÉCUTION** à **PASSIVE**, le **gestionnaire compare** les priorités des tâches qui sont dans l'état **PRÊTE** et produit l'événement *EXÉCUTER* pour l'automate de la tâche la plus prioritaire qui passe de l'état **PRÊTE** à **EXÉCUTION**. Si cette tâche a été préemptée le **gestionnaire** restaure son contexte et il lance son exécution sinon il lance son exécution.

Le **gestionnaire** produit l'événement *BLOQUER* lorsqu'une tâche ne peut pas accéder à une ressource partagée déjà utilisée par une autre tâche. Elle passe alors de l'état **EXÉCUTION** à l'état **BLOQUÉE**. Il produit l'événement *DÉBLOQUER* lorsqu'elle peut de nouveau y accéder. Elle passe alors de l'état **BLOQUÉE** à l'état **PRÊTE**.

Ordonnancement temps réel monoprocesseur

En ligne préemptif 6/10 : faisabilité, ordonnançabilité, conditions d'ordonnançabilité

L'analyse de faisabilité d'un ensemble de tâches temps réel consiste à trouver des conditions où toutes les tâches **satisfont** leurs contraintes.
L'analyse d'ordonnançabilité consiste à trouver de telles conditions mais avec un algorithme d'ordonnancement **donné**.
Pour un ensemble de n tâches **préemptives périodiques indépendantes** dont le coût de l'ordonnanceur et de la préemption est **approximé** dans le WCET, le facteur d'utilisation est $U = \sum_{i=1}^{n} C_i/T_i$ et
la densité est $\Delta = \sum_{i=1}^{n} C_i/D_i$.
Voici leur **condition d'ordonnançabilité** suffisante et nécessaire et suffisante avec les **algorithmes d'ordonnancement** suivants :

- à priorités fixes (ne changent pas pendant l'exécution des tâches) :
  - algorithme **RM** (Rate Monotonic) : priorités des tâches inversement proportionnelle à leur période, **il suffit que** $U \leq n(2^{1/n} - 1)$, $D_i = T_i$, pour que les tâches soient ordonnançables ;
  - algorithme **DM** (Deadline Monotonic) : priorités des tâches inversement proportionnelle à leur échéance relative, **il suffit que** $\Delta \leq n(2^{1/n} - 1)$, $D_i \leq T_i$, pour que les tâches soient ordonnançables ;

Ordonnancement temps réel monoprocesseur

En ligne préemptif 7/10 : conditions d'ordonnançabilité, optimalité

- à priorités dynamiques (calculées à chaque activation et terminaison) :
  - algorithme **EDF** (Earliest Deadline First) : priorité à la tâche de plus petite échéance absolue, *fixe dans l'instance, dynamique entre instances*,
    il faut et il suffit que $U \leq 1$, $D_i = T_i$, pour que les tâches soient ordonnançables,
    il suffit que $\Delta \leq 1$, $D_i \leq T_i$, pour que les tâches soient ordonnançables ;
  - algorithme **LLF** (Least Laxity First) : priorité à la tâche de plus petite laxité, *dynamique dans l'instance, dynamique entre instances*,
    mêmes conditions que EDF pour que les tâches soient ordonnançables.

Un algorithme d'ordonnancement temps réel est dit **optimal** s'il trouve une solution au problème d'ordonnancement lorsqu'il en existe une. Si un algorithme optimal ne trouve pas de solution au problème, elle n'existe pas.
On a par exemple les résultats d'optimalité temps réel suivants :
- **RM** optimal pour des tâches préemptives périodiques ou sporadiques pour $D_i = T_i$ et non optimal pour $D_i \neq T_i$;
- **RM** pas optimal pour des tâches non préemptives ;
- **EDF** optimal pour des tâches préemptives ;
- **EDF** optimal pour des tâches non préemptives si r_i^1 non imposées, sinon non optimal.

Ordonnancement temps réel monoprocesseur

En ligne préemptif 8/10 : tâches dépendantes 1/2

Lorsque les **tâches sont dépendantes**, cela est dû soit à :

- des **précédences seulement**, on peut alors se ramener au cas sans dépendances en modifiant les dates d'activation et les échéances des tâches de manière à ce qu'une tâche ait sa date d'activation plus petite que celles de ses prédécesseurs et que son échéance soit plus petite que celles de ses successeurs ;

- des **dépendances de données**, en plus de la contrainte de précédence il faut gérer d'une part le **partage des données** entre tâches productrices et tâches consommatrices et d'autre part le **transfert des données** en fonction de la valeur (plus petite ou plus grande) de la période de la tâche productrice T_p par rapport à celle de la tâche consommatrice T_c. Le nombre maximum de données que doit attendre la tâche consommatrice est : $n = \lceil \frac{T_c}{T_p} \rceil$.
  Si $T_c > T_p$ alors T_p produit $n > 1$ données que T_c consomme une fois, sinon T_p produit 1 donnée et T_c consomme n fois cette même donnée.

  En monoprocesseur le coût du transfert des données est souvent négligé car il correspond à un accès mémoire. En multiprocesseur **il ne peut pas être négligé dans le cas de communications par passage de messages**.

Ordonnancement temps réel monoprocesseur

En ligne préemptif 9/10 : tâches dépendantes 2/2

Si une **donnée est partagée** par plusieurs tâches dépendantes, une tâche qui utilise la donnée **protégée par une section critique** peut ne pas pouvoir être préemptée par une tâche plus prioritaire voulant accéder à cette donnée, ce qui conduit à un **problème d'inversion de priorité**.
L'automate d'une tâche qui veut accéder à une donnée passe dans l'état **BLOQUEE** tant que celle-ci n'est pas disponible. Un **problème d'interblocage** peut se produire si plusieurs tâches sont dans cet état.
Les deux **protocoles** pour éviter, dans l'ordre, ces deux problèmes sont :

- l'**héritage de priorité** (priority inheritance protocol PIP) : la tâche s'exécutant en accédant à une donnée hérite de la plus haute priorité des tâches qui partagent cette donnée afin qu'elle la libère au plus tôt, on peut alors calculer son temps de blocage maximum ;
- la **priorité plafonnée** (priority ceiling protocol PCP) : afin d'éviter les interblocages, le protocole précédent est étendu en ajoutant une priorité à chaque donnée, égale à la plus haute priorité (plafond) des tâches qui la partagent, ainsi une tâche ne peut accéder à une donnée que si la priorité de cette donnée est supérieures aux priorités des autres données auxquelles accèdent les autres tâches.

Ordonnancement temps réel monoprocesseur

En ligne préemptif 10/10 : tâches apériodiques

Pour un ensemble de **tâches apériodiques** les algorithmes EDF et LLF peuvent être utilisés tels quels puisque la priorité dynamique des tâches est calculée lors de l'exécution relativement à l'échéance absolue ou la laxité.

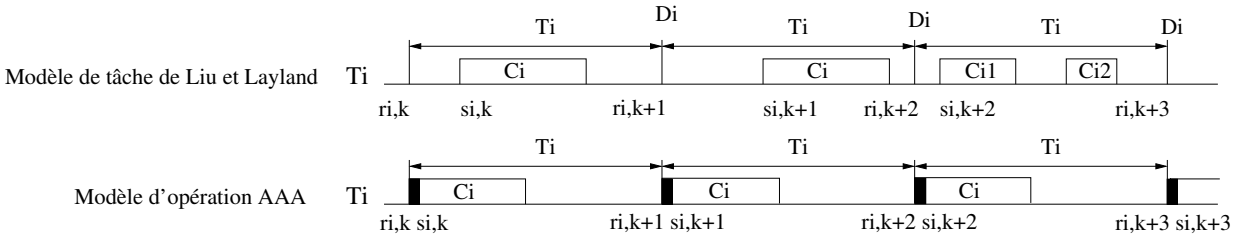
D'autres algorithmes d'ordonnancement plus spécifiques qui peuvent être fondés sur des priorités fixes, ont été proposés :

- **arrière plan** : elles sont ordonnancées quand le processeur n'ordonnance pas d'autres tâches périodiques ;
- **serveur de tâches apériodiques** : elles sont exécutées par une tâche périodique supplémentaire qui a une certaine capacité :
  - **serveur à scrutation** : la capacité est remise à zéro si pas de tâches apériodiques en attente ou la capacité est entièrement utilisée dans l'instance ;
  - **serveur ajournable** : la capacité est remise à zéro quand la capacité est entièrement utilisée dans l'instance.
- **vol de marge** : elles sont ordonnancées pendant les laxités des autres tâches périodiques.

Ordonnancement temps réel monoprocesseur

Hors ligne non préemptif AAA 1/2

Les applications temps réel **critiques** utilisent un ordonnanceur **hors ligne**. Elles ont des tâches traitant les capteurs (resp. les actionneurs) qui doivent être sans gigue. Elles doivent donc avoir, une **période stricte**, c.a.d. que leurs dates de début d'exécution doivent être égales à leurs dates d'activation plus le **coût de l'exécutif constant** (resp. être **non préemptives** afin que leur temps de réponse ne varie pas, car égal à leur WCET). Pour simplifier le problème on considère que toutes les tâches sont de ce type. Elles suivent le ≪ **modèle d'opération AAA** ≫ où une **opération** (tâche) se répète infiniment avec une **période stricte** T_i et un pire temps d'exécution C_i augmenté du coût constant de l'exécutif α. On a $s_i^k = r_i^k + \alpha$ pour chaque instance k et donc $s_i^{k+1} = s_i^k + T_i = s_i^1 + k\,T_i$. Son **échéance relative est égale à la période** imposant que $C_i \leq T_i$.



Ordonnancement temps réel monoprocesseur

Hors ligne non préemptif AAA 2/2

Les opérations apériodiques sont **rendues périodiques** en interrogeant les événements qui les déclenchent à une période plus petite que le délai minimum entre deux de leurs apparitions. On cherche donc à résoudre un problème d'ordonnancement monoprocesseur **non préemptif** d'un ensemble d'opérations (tâches) devant, chacune, respecter ses contraintes de dépendance et **d'échéance égale à sa période stricte**.

Analyse de faisabilité hors ligne non préemptif

Elle est fondée sur la **condition suffisante** suivante (Korst 1991 pour deux tâches et Kermia, Sorel 2009 pour plus de deux tâches) : pour un graphe de n tâches non préemptives dépendantes de durées d'exécution C_i et de périodes strictes T_i, **il suffit que** $\sum_{i=1}^{n} C_i \leq PGCD(T_i)$, pour que les tâches soient ordonnançables. Elle est uniquement réalisée sur un intervalle de temps de durée égale à $PPCM(T_i)$ qui se répétera infiniment. Elle produit une **table d'ordonnancement** contenant, sur cet intervalle, la date de début d'exécution et la durée pendant laquelle chaque instance de tâche doit s'exécuter.

Ordonnancement temps réel monoprocesseur

Hors ligne préemptif AAA 1/4

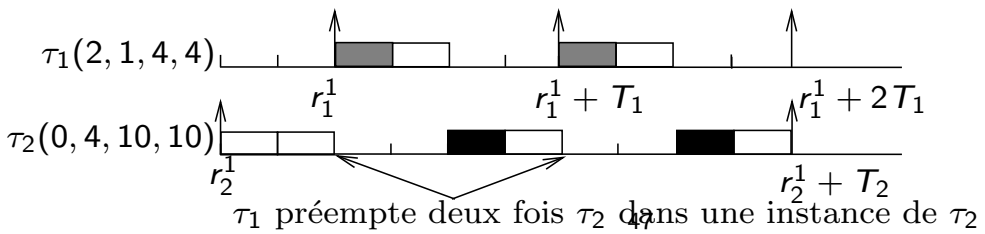
Quand le facteur d'utilisation est proche de 1 et/ou que les tâches (opérations) de grandes périodes relativement aux autres tâches, ont des WCET qui sont eux-aussi grands, pour **augmenter** le taux d'ordonnançabilité il faut utiliser la préemption. **Il faut alors maîtriser précisément son coût**. Dans ce but on fait, pour un algorithme d'ordonnancement donné (RM, DM, EDF), une **analyse d'ordonnançabilité hors ligne** tenant compte du coût constant de l'exécutif α, sans l'approximer dans les WCET des tâches comme cela est fait dans les analyses d'ordonnançabilité en ligne. Elle est uniquement réalisée sur un intervalle de temps dont la durée est $I_n = [r_{min}, r_{max} + 2PPCM(T_i)]$ et dont une partie se répétera infiniment, avec r_{min} la valeur minimale des premières dates d'activations des tâches, r_{max} la valeur maximale et T_i la période de chacune des n tâches. Elle produit une **table d'ordonnancement** contenant, sur cet intervalle, la date de début d'exécution, ou de reprise si elle est préemptée et la durée pendant laquelle chaque instance de tâche doit s'exécuter.

Pour éviter les pertes de données les **tâches dépendantes ont des périodes multiples**. De plus, toutes les données produites sont consommées afin d'être dans le pire des cas. On recalcule les dates d'activation et d'échéance des tâches dépendantes, et on calcule le nombre de données que les tâches doivent produire et consommer. Pour éviter les interblocages on applique le **protocole de la priorité plafonnée (PCP)**.

Ordonnancement temps réel monoprocesseur

Hors ligne préemptif AAA 2/4

Le coût de l'exécutif est formé : 1) d'un coût payé par chaque **tâche qui est activée et éventuellement préempte** (gris) une autre tâche, correspondant au choix de la prochaine tâche à exécuter en lisant dans la table d'ordonnancement et à la sauvegarde/restauration de contexte, 2) d'un coût payé par chaque **tâche préemptée** (noir) correspondant au choix de la prochaine tâche à exécuter en lisant dans la table d'ordonnancement et à la sauvegarde/restauration de contexte. Une tâche qui préempte paye le coût (1) à chaque instance alors qu'une tâche préemptée peut payer plusieurs fois le coût (2) dans une instance en fonction du nombre de fois où elle est préemptée. Cette approche permet de tenir compte des préemptions causées par **d'autres préemptions**. Ici les deux coûts valent 1 unité pour mieux les visuliser.



τ_1 préempte deux fois τ_2 dans une instance de τ_2

Ordonnancement temps réel monoprocesseur

Hors ligne préemptif AAA 3/4

Analyse d'ordonnançabilité hors ligne préemptif

Initialiser l'ensemble des **dates d'appel de l'ordonnanceur** t avec les **dates d'activation** des tâches dans l'intervalle I_n, $t \leftarrow r_{min}$, $ordonnançable = V$

Tant que $t \leq r_{max} + 2H_n \wedge (ordonnançable = V)$

sélectionner la plus prioritaire parmi les tâches prêtes en utilisant un algorithme d'ordonnancement et en appliquant PCP, calculer son temps restant à exécuter $c_i(t)$ en fonction de la tâche précédente sélectionnée et, **en tenant compte du coût de l'exécutif** α, calculer l'échéance relative $d_i(t)$

Si $c_i(t) > d_i(t)$ **Alors** $ordonnançable \leftarrow F$ **Fin si**

$t \leftarrow$ appel suivant : prochaine activation connue ou terminaison calculée

Si fin tâche plus petite que t **Alors** début de tâche inactive jusqu'à t **Fin si**

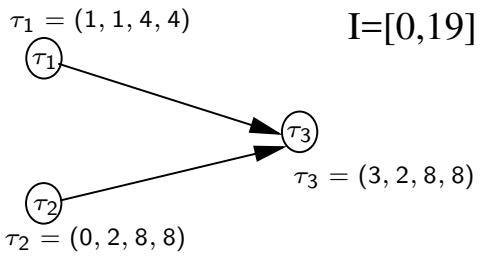
Si $ordonnançable = V$ **Alors** créer une ligne dans la table d'ordonnancement

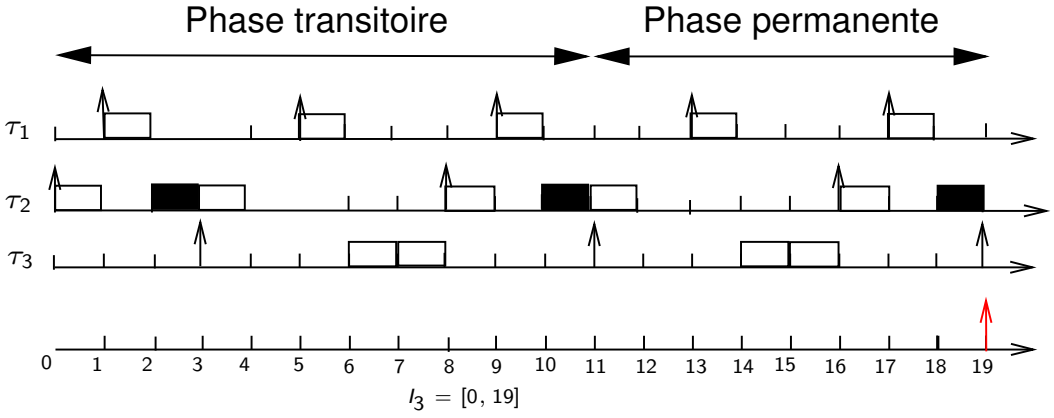
Fin tant que

Chaque ligne de la table d'ordonnancement contient 4 colonnes : ***t*** la date d'appel de l'ordonnanceur, $\boldsymbol{\tau_i}$ le nom de la tâche à exécuter, ***e*** la durée pendant laquelle elle doit s'exécuter, ***s*** son statut indiquant : ***d*** si elle débute, ***c*** si elle continue, ***r*** si elle reprend après avoir été préemptée, ou bien le nom de tâche ≪ *inactive* ≫ si elle n'a rien à faire jusqu'au prochain appel de l'ordonnanceur.

Ordonnancement temps réel monoprocesseur

Hors ligne préemptif AAA 4/4 algorithme d'ordonnancement RM





Statut s : d = début, c = continuation, r = reprise

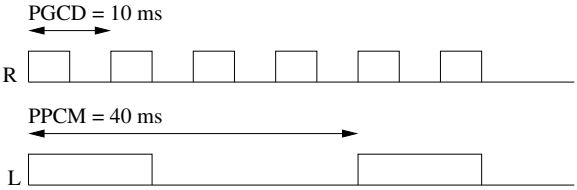
Table d'ordonnancement

t	$\boldsymbol{\tau_i}$	***e***	***s***
0	τ_2	1	*d*
1	τ_1	1	*d*
2	τ_2	1	*r*
3	τ_2	1	*c*
4	*inactive*		
5	τ_1	1	*d*
6	τ_3	2	*d*
8	τ_2	1	*d*
9	τ_1	1	*d*
10	τ_2	1	*r*
11	τ_2	1	*c*
12	*inactive*		
13	τ_1	1	*d*
14	τ_3	2	*d*
16	τ_2	1	*d*
17	τ_1	1	*d*
18	τ_2	1	*r*
19	τ_2	1	*c*

Langages synchrones et ordonnancement temps réel

Hors ligne

- Cas mono-horloge avec et sans sous-échantillonnage : il faut exécuter l'ensemble des instructions Signal entre deux instants logiques qui correspondent alors à la période de l'horloge physique qu'il faut choisir de manière à ce qu'elle soit supérieure ou égale au temps d'exécution total des instructions Signal (opérations) de l'algorithme sur l'architecture monprocesseur.
- Cas multi-horloge : pour chaque signal d'entrée on choisit une horloge physique telle que les périodes des horloges des signaux d'entrée soient des multiples les unes des autres. Cela conduit à des instructions Signal (tâches) dont les périodes sont aussi des multiples.



Si le WCET de la tâche la plus lente est plus petit que la laxité de la tâche la plus rapide, la plus lente n'est pas nécessairement préemptée, sinon la plus lente doit être préemptée pour être ordonnançable.

Langages synchrones et ordonnancement temps réel

En ligne

On construit un système de tâches à échéances sur requête ayant la même période égale au PGCD des périodes des tâches. Si elles sont premières on prend la plus petite.

On utilise un ordonnanceur en ligne appliquant un algorithme d'ordonnancement à priorités fixes, qui dans ce cas sont toutes les mêmes. Les tâches ont alors la même date d'activation et doivent se terminer avant l'activation suivante.

Les périodes T_i des tâches étant des multiples de leur PGCD on a $T_i = n_i PGCD$. Comme toutes les tâches sont activées en même temps, à chaque appel de l'ordonnanceur il suffit d'exécuter celle correspondant au nombre de fois $PPCM/n_i PGCD$ qu'elle doit se répéter relativement aux valeurs que prend un compteur modulo $PPCM/PGCD$.

Ordonnancement temps réel multiprocesseur

En ligne 1/2

Il y a 2 approches principales pour ordonnancer en multiprocesseur :

- **global :** un ordonnanceur unique pour l'ensemble des processeurs qui peut faire **migrer** les tâches d'un processeur à l'autre, ce qui a un coût très élevé et variable, donc pas adapté au temps réel critique ;
- **partitionné :** un ordonnanceur par processeur sur chacun desquels on a distribué ou alloué (partitionné) les tâches. Allouer des tâches sur un nombre de processeurs fixé ou en cherchant à minimiser ce nombre, avec comme contrainte que sur chaque processeur les tâches allouées sont faisables ou ordonnançables, et éventuellement en minimisant la latence, est un problème **NP-difficile** équivalent à un problème de remplissage de boîtes à une dimension, de nombre fixe ou à minimiser, avec des objets de tailles différentes (« Bin Packing »). Il ne peut se résoudre en un temps acceptable que de manière approchée (non optimale) avec des **heuristiques**.
- **semi-partitionné** : ordonnanceur partitionné à migrations minimisées ;
- **par groupe (clustering)** : migration uniquement sur certains processeurs.

Les communications sont en général traitées à part, souvent sans considérer leur coût.

Ordonnancement temps réel multiprocesseur

En ligne 2/2

- **Global :** on peut utiliser n'importe quel algorithme à priorité fixe ou dynamique mais ce n'est pas optimal. L'algorithme « Pfair » (Proportionate Fair) est optimal. Il consiste à découper chaque tâche en sous-tâches de durée unitaire et à les ordonnancer équitablement sur l'ensemble des processeurs. Cette solution théorique, négligeant le coût de migration, utilise des pseudos, activations et échéances, pour chaque sous-tâche. Elle conduit à différents algorithmes comme EPDF (Earliest Pseudo-Deadline First), DPF (Deadline Partitioning Fair), etc.
- **Partitionné :** on utilise les heuristiques classiques suivantes :
  - **NF Next Fit :** distribuer chaque tâche de la liste des tâches sur le premier processeur de la liste tant que **les tâches** sont ordonnançables, puis les distribuer sur le processeur suivant (next), puis continuer ;
  - **FF First Fit :** distribuer chaque tâche de la liste des tâches sur le premier processeur pour lequel **la tâche** est ordonnançable (first), **recommencer au début** de la liste des processeurs, puis continuer ;
  - **BF Best Fit :** distribuer chaque tâche sur le processeur pour lequel les tâches sont ordonnançables et critère le plus petit, puis continuer ;
  - **WF Worst Fit :** distribuer chaque tâche sur le processeur pour lequel les tâches sont ordonnançables et critère le plus grand, puis continuer.

Ordonnancement temps réel multiprocesseur

AAA hors ligne

A cause du coût prohibitif de la migration, on se focalise sur l'**ordonnancement partitionné** par la suite. Il faut donc résoudre un problème de distribution sur les différents processeurs et d'ordonnancement monoprocesseur **hors ligne**, **avec** ou **sans préemption**, tenant compte de son **coût**, d'un ensemble d'opérations (tâches) devant respecter des **contraintes de dépendance** et **d'échéances**. De plus on cherche à minimiser le temps d'exécution total en **considérant les temps de communication inter-processeur** dues aux dépendances.

Réaliser une distribution et un ordonnancement consiste à **transformer** le graphe d'algorithme **en fonction** du graphe d'architecture dans lequel on a déterminé toutes les routes possibles. Cela revient à réduire le **parallélisme potentiel** de l'algorithme (ordonnancement) pour le faire correspondre au **parallélisme effectif** de l'architecture (distribution).

Une implantation **optimisée** est obtenue en minimisant le temps d'exécution total noté « **latence de l'application** » = Max(latences E/S).

Formalisation de l'implantation distribuée

Transformations du graphe d'algorithme en fonction du graphe d'architecture

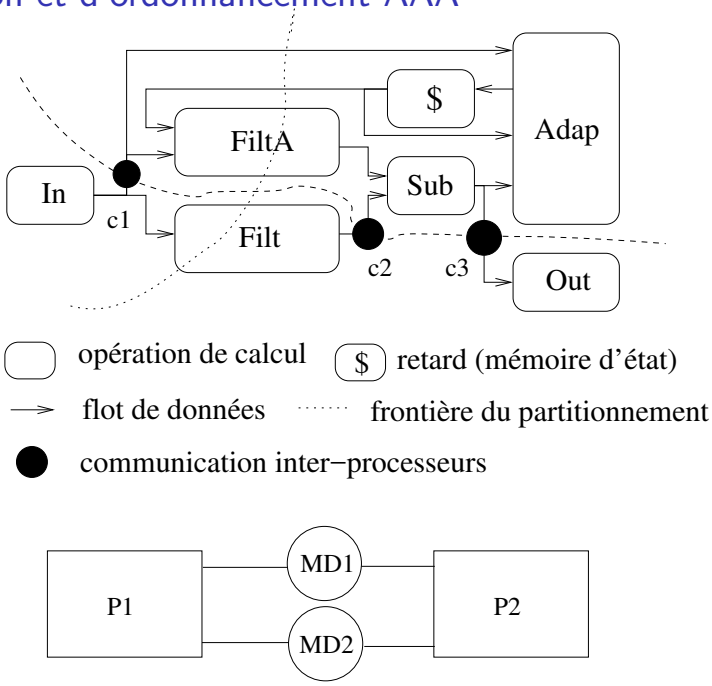
La **formalisation** décrit la distribution et l'ordonnancement des opérations (tâches) uniquement du point de vue des dépendances **sans considérer leurs caractéristiques et contraintes temporelles** qui seront exploitées lors de l'optimisation.

Le graphe d'algorithme est transformé de la manière suivante :

- **faire une partition** de l'ensemble des opérations en autant d'éléments qu'il y a d'opérateurs dans l'architecture ;
- **remplacer** les arcs reliant les éléments de partition par autant de **nouveaux sommets de communication et d'arcs** qu'il y a de média dans la route sur laquelle ces opérations de communication sont distribuées ;
- **ajouter** des arcs de précédence entre les opérations non dépendantes distribuées sur un même processeur ;
- **ajouter** des arcs de précédence entre opérations de communications non dépendantes distribuées sur un même médium.

Formalisation de l'implantation distribuée

Exemple de distribution et d'ordonnancement AAA



{In, Filt, Out} distribuées sur P1 et {FiltA, sub, Adap} distribuées sur P2. {c1} distribuée sur COM11/SAM11/MD1/SAM21/COM21 et {c2, c3} distribuées sur COM12/SAM12/MD2/SAM22/COM22, communications par passage de message via des média point-à-point.

Formalisation de l'implantation distribuée

Principes

L'implantation (distribution et ordonnancement) est une transformation de graphes **formalisée par la composition de trois relations** chacune d'elles s'appliquant sur deux couples de graphes (algorithme, architecture) :

- **routage** : connexion complète du graphe de l'architecture ;
- **distribution** des opérations de l'algorithme sur les opérateurs ;
- **distribution** des opérations de communication induites par la distribution précédente sur les média ;
- **ordonnancement** des opérations sur les opérateurs où elles ont été distribuées et des opérations de communications, issues des dépendances reliant des opérations distribuées sur des opérateurs différents, sur les média où elles ont été distribuées.

Le nombre de distributions et le nombre d'ordonnancements que l'on peut réaliser à partir d'un algorithme et d'une architecture donnés **peut être très grand mais il est calculable**. Cette composition de relations **doit conserver les propriétés temporelles logiques** garanties lors des vérifications formelles effectuées sur la spécification fonctionnelle.

Routage

Détermination de tous les chemins. Un chemin (route) dans le graphe d'architecture est une suite de sommets reliés par des arcs, formant un ordre total.

P = ensemble des processeurs, $Card(P) = p$
L = ensemble des média de communication, $Card(L) = l$
X = ensemble des connexions, $x = (p, l)$ ou (l, p), $p \in P$ et $l \in L$,
R = ensemble des chemins de $(P \cup L, X)$, $r = (p, l, p', l', p''), p, p', p'' \in P$ et $l, l' \in L$

$$(P \cup L, X) \stackrel{routage}{\longrightarrow} (P \cup L, R)$$

Distribution

O = ensemble des opérations, $Card(O) = n$
D = ensemble des dépendances, $d = (o, o'), o, o' \in O$

Distribution des opérations sur les opérateurs = partition des n opérations de O en p éléments, $n > p$. Le nombre de partitions possibles est **calculable** égal à : $\sum_{k=0}^{p} (-1)^k \frac{(p-k)^n}{(p-k)!k!}$
Par exemple pour $n = 4$, $p = 2$ on a 7 partitions possibles, pour $n = 12$, $p = 3$ on en a 86 526 et pour $n = 12$, $p = 5$ on en a 1 379 400.

$O \supset O_p$ = ensemble des opérations exécutées par le processeur p
$D \supset D_p$ = ensemble des dépendances entre opérations exécutées par p
$D \supset D_r$ = ensemble des dépendances inter-partition

$$((O, D), (P \cup L, R)) \stackrel{distrib}{\longrightarrow} (G_{dR}, (P \cup L, R)) \quad G_{dR} = \left(\bigcup_{p \in P} (O_p, D_p), D_r \right)$$

Formalisation de l'implantation distribuée

Relation de distribution 2/2

Distribution des opérations de communication sur les média = partition de D_r en l éléments dont le nombre est pareillement **calculable**.

Chaque dépendance inter-partition $(o_{i\,p_i}, o_{j\,p_j})$, avec o_i sur p_i, o_j sur p_j, est transformée en un chemin (ordre total) comportant un sommet pour chacun des m média de la route sur laquelle elle est distribuée :

$\forall r \in R, \forall d_r \in D_r \quad d_r \xrightarrow{com} \left(o_{i_{p_i}}, o_{l_1}, o_{l_2}, \ldots o_{l_{k-1}}, o_{l_k}, \ldots o_{l_m}, o_{j_{p_j}}\right)$

Un sommet o_{l_k} est une nouvelle **opération de communication** distribuée sur le médium l_k. Un arc $(o_{l_{k-1}}, o_{l_k}) = c_p$ est une dépendance distribuée sur le processeur p.

On regroupe les o_{l_k} d'un même $l_k \in L$ dans l'ensemble O_l et les c_p d'un même $p \in P$ dans l'ensemble C_p avec $C_p = C_{p(calc,com)} \cup C_{p(com,com)} \cup C_{p(com,calc)}$

$$G_{dR} \xrightarrow{com} G_{dL} = \left(\bigcup_{p \in P}(O_p, D_p \cup C_p), \bigcup_{l \in L} O_l\right)$$

Formalisation de l'implantation distribuée

Relation d'ordonnancement

L'ajout des opérations de communication o_l et leur distribution sur les média **ne modifient pas l'ordre partiel** D du graphe algorithme. Le nombre de sommets et d'arcs augmente en fonction du nombre de média.

Ordonnancement

Sur chaque processeur p, on réalise un ordonnancement des opérations de calcul qui est un ordre total $\bar{D}_p$ incluant l'ordre partiel D_p, $D_p \subseteq \bar{D}_p$, obtenu par ajout de **précédences seulement**. De même sur chaque médium l, on réalise un ordonnancement des opérations de communication qui est un ordre total $\bar{D}_l$ incluant l'ordre partiel D_l, $D_l \subseteq \bar{D}_l$, obtenu par ajout de **précédences seulement**.

$$G_{dL} \xrightarrow{sched} G_s = \left(\bigcup_{p \in P}(O_p, \bar{D}_p \cup\ C_p), \bigcup_{l \in L}(O_l, \bar{D}_l)\right)$$

Le nombre d'arcs augmente en fonction des opérations non dépendantes. Le nombre d'ordres totaux tirés d'un ordre partiel est **calculable** égal à $C_n^2 = \dfrac{n!}{2!(n-2)!}$ pour n opérations non dépendantes. On a $C_{10}^2 = 45$.

Formalisation de l'implantation distribuée

Composition de trois relations

Implantation = Routage o Distribution o Ordonnancement

L'implantation (distribution et ordonnancement) est la transformation de graphes *dist/sched* formalisée par la composition des relations :

$$((O, D), (P \cup L, X)) \overset{routage}{\longrightarrow} ((O, D), (P \cup L, R))$$

$$((O, D), (P \cup L, R)) \overset{distrib}{\longrightarrow} (G_{dR}, (P \cup L, R)) \overset{com}{\longrightarrow} (G_{dL}, (P \cup L, R)) \overset{sched}{\longrightarrow} (G_s, (P \cup L, R))$$

$$((O, D), (P \cup L, R)) \overset{dist/sched}{\longrightarrow} (G_s, (P \cup L, R))$$

Puisque le nombre de partitions (NP_{calc}, NP_{com}) et le nombre d'ordres totaux NO tirés d'un ordre partiel sont calculables, alors **le nombre d'implantations possibles est calculable** $= NP_{calc} NP_{com} NO$.

Formalisation de l'implantation distribuée

Loi de composition externe

Si on suppose que l'on a une architecture où toutes les routes sont connues, cette composition de trois relations peut aussi se voir comme une **loi de composition externe** notée $\oplus$. Soit G_{al} l'ensemble des graphes d'algorithme et G_{ar} l'ensemble des graphes d'architecture, on a alors :

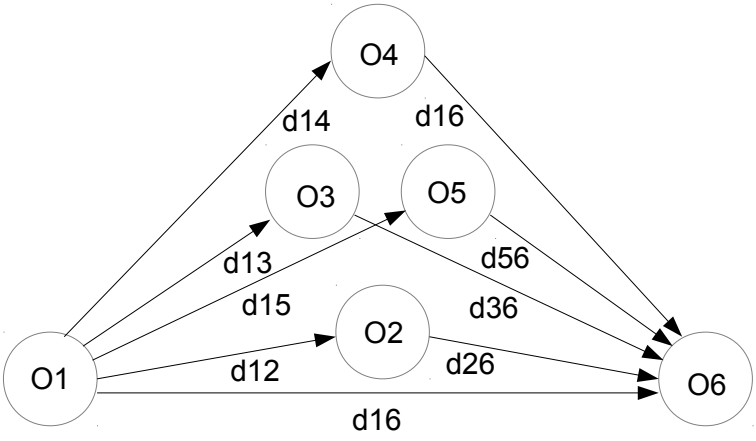
$$G_{al} \times G_{ar} \longrightarrow G_{al} \qquad g_{al} \oplus g_{ar} = g'_{al}$$

On choisit, parmi toutes les transformations de graphes, celles qui correspondent à des **implantations valides** pour lesquelles l'ordre partiel obtenu est **compatible** avec l'ordre partiel initial du graphe d'algorithme, c.a.d. qu'aucun sommet ou arc n'a été supprimé, que seulement des arcs de précédence ont été ajoutés, et finalement que des sommets et des arcs de précédence ont été ajoutés sous la forme de chemins (ordre total), uniquement, pour remplacer des arcs inter-partition.

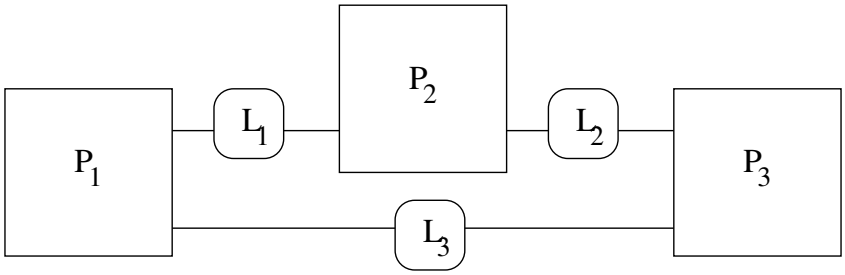
Les **arcs de précédence** (pas de transfert de données) ajoutés lors de l'ordonnancement entre les opérations de calcul qui n'avaient pas de dépendances **ne peuvent pas créer de cycle de dépendances de donnée**, idem pour les **arcs de précédence** ajoutés entre les opérations de communication.

Formalisation de l'implantation distribuée

Exemple de formalisation d'implantation AAA 1/3



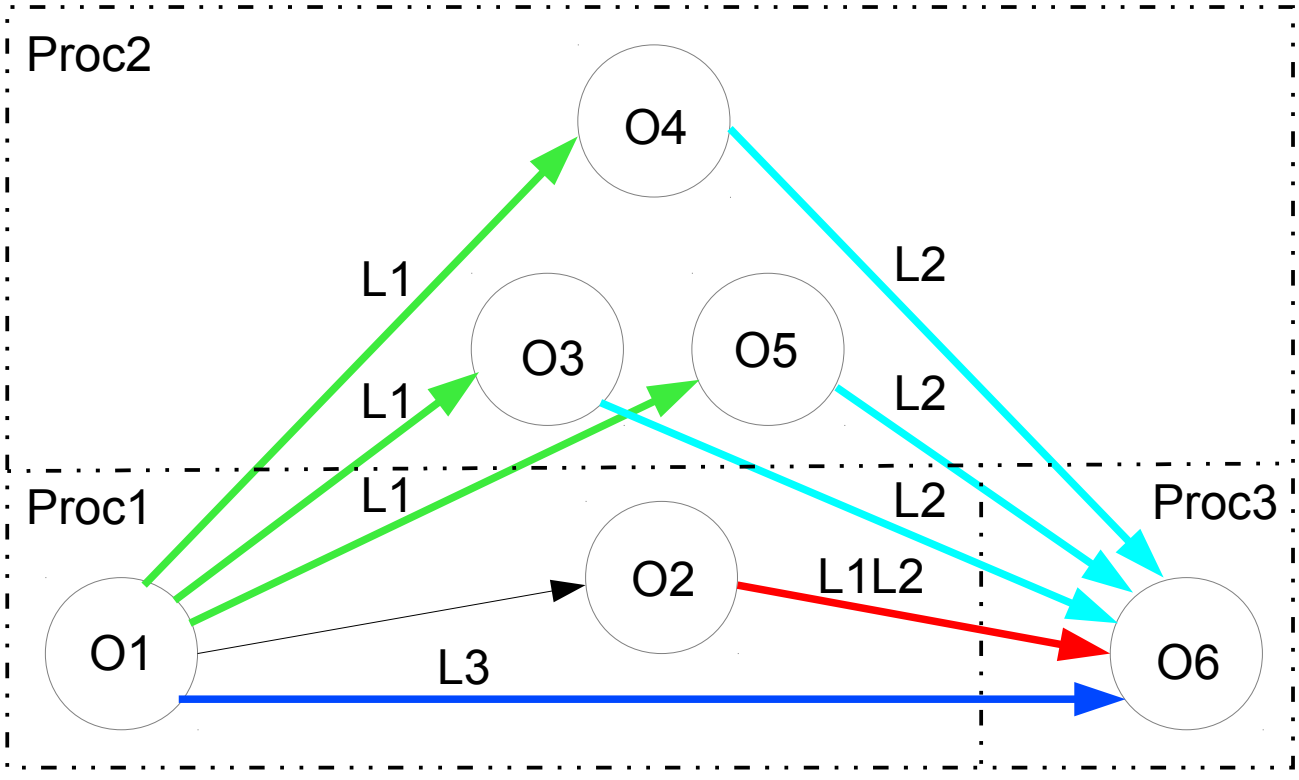
Graphe d'algorithme



Graphe d'architecture

Formalisation de l'implantation distribuée

Exemple de formalisation d'implantation AAA 2/3

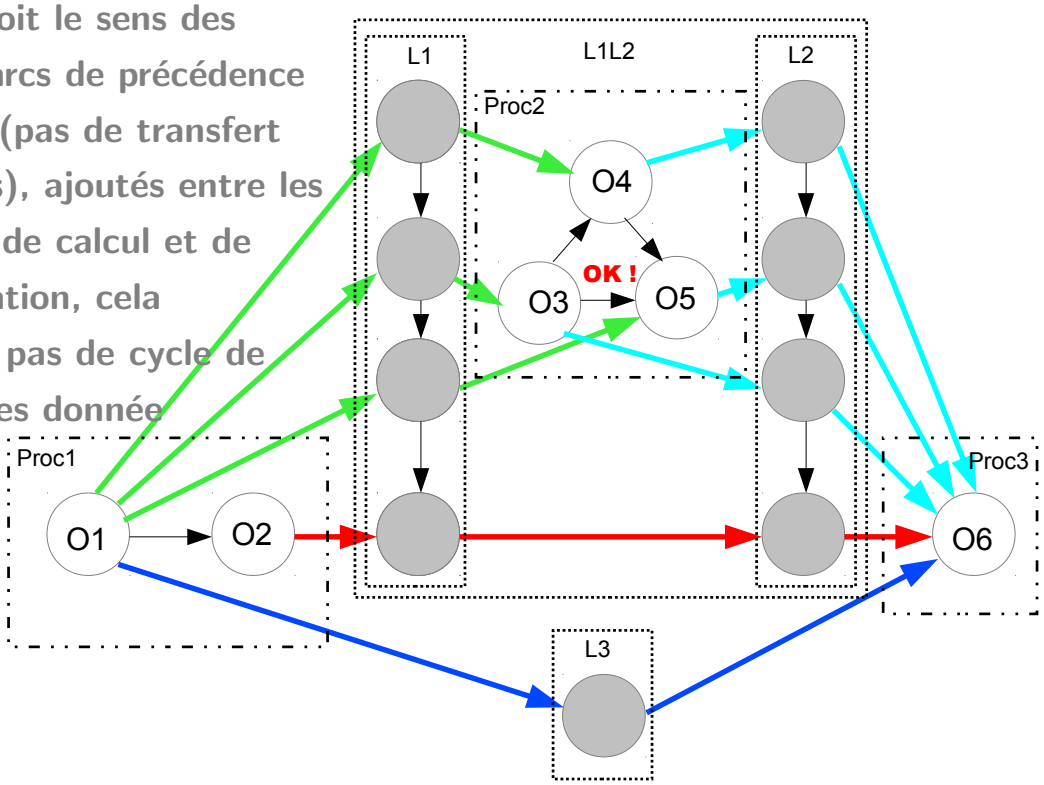


Distribution des opérations et des dépendances

Formalisation de l'implantation distribuée

Exemple d'implantation AAA 3/3

Quel que soit le sens des nouveaux arcs de précédence seulement (pas de transfert de données), ajoutés entre les opérations de calcul et de communication, cela n'introduit pas de cycle de dépendances donnée



Ordonnancement des opérations de calcul et de communication.

Optimisation de l'implantation distribuée

Principes 1/3

Parmi un nombre d'implantations valides qui peut être très grand, il s'agit de rechercher, soit manuellement soit automatiquement, une **implantation optimisée** appelée **adéquation**.

Le problème d'optimisation automatique va consister d'abord à choisir, parmi toutes les implantations valides, celles qui pour chaque opération respectent ses contraintes de **dépendance** et d'**échéance égale à sa période stricte** et ensuite à **minimiser la latence** de l'algorithme sur l'architecture. De plus on pourra, par exemple, essayer de **minimiser le nombre de composants de l'architecture**, etc.

Pour faire un choix parmi les implantations valides on doit avoir caractérisé chaque opération et chaque dépendance de l'algorithme **en terme de durée d'exécution** relativement à l'architecture et éventuellement avoir caractérisé chaque opération en terme de **période** et donc d'**échéance** (égale à la période) si elle en a une. Ceci permet d'obtenir un graphe d'algorithme étiqueté par ces caractéristiques.

Optimisation de l'implantation distribuée

Principes 2/3

Le problème d'implantation **optimale**, selon l'approche partitionnée, est équivalent à un problème de « Bin Packing ». Ce dernier appartient à la **classe de complexité NP-difficile** qui contient les problèmes plus difficiles que ceux de la **classe NP-complet** qui, elle, contient les problèmes les plus difficiles de la **classe NP**. Cette dernière contient les problèmes pouvant être résolus sur une **machine de Turing non déterministe** en temps polynomial relativement à la taille du problème, donc **NP** vient de « **N**on déterministe en temps **P**olynomial ». Ce sont des problèmes pouvant être résolus en énumérant toutes les solutions, chacune pouvant être testée en temps polynomial. Ils sont résolus optimalement en **temps exponentiel**. La **classe P** contient les problèmes pouvant être résolus sur une **machine de Turing déterministe** en temps polynomial. Déterministe (resp. non déterministe) veut dire que depuis un état quelconque de la machine de Turing, il y a une seule (resp. plusieurs) transition possible.

Pour les problèmes de petites tailles, on peut trouver des solutions optimales en utilisant la programmation linéaire en nombres entiers, dynamique ou par contraintes, les algorithmes d'évaluation séparation (B & B, B & C, etc.), etc.

Optimisation de l'implantation distribuée

Principes 3/3

Pour les problèmes de tailles réalistes, on utilise des **heuristiques** qui trouvent des solutions empiriquement proches de la solution optimale. Les **algorithmes d'approximation** trouvent des solutions proches de la solution optimale à un facteur ϵ près. Il y a deux types d'heuristiques :

- **sans retour arrière** : très rapides, dites « **gloutonnes** ». Elles cherchent à chaque étape une solution localement optimale, sans remettre en cause les solutions obtenues précédemment, pour construire une solution finale qui n'est pas en général globalement optimale. Elles font des choix déterministes en général dans une liste. Elles sont adaptées à des problèmes spécifiques ;
- **avec retour arrière** : moins rapides, dites de « **recherche locale** ». Elles partent d'une solution initiale qu'elles transforment itérativement en vue de l'améliorer en faisant une recherche dans le « **voisinage** » de la solution courante. Elles peuvent faire des choix aléatoires pour sortir de minima locaux. Elles donnent des résultats empiriquement plus proches de la solution optimale que les heuristiques gloutonnes, mais en un temps plus long. Comme elles résolvent des problèmes génériques on les appelle « **métaheuristiques** », comme par exemple : « recuit simulé », « recherche tabou », « algorithme génétique », « colonie de fourmis », etc.

Optimisation de l'implantation distribuée

Caractérisation des opérations et des dépendances

Chaque opération **est caractérisée relativement aux opérateurs** qui sont capables de l'exécuter. Chaque dépendance est **caractérisée relativement aux média** qui sont capables de l'exécuter.

- **opérateur** et **communicateur**
  nom d'opération → durée d'exécution (mesurée hors arbitrage)
  données transférée → durée d'exécution (mesurée hors arbitrage)

ASICfft	
fft256	15

C40alu	
fft256	1250
mul10	14
add10	14

C40dma	
logical	9=3+6
integer	9=3+6
[10]real	63=3+10*6

- **mux** (arbitrage) ralentissement opérateur1 lorsque opérateur2 actif

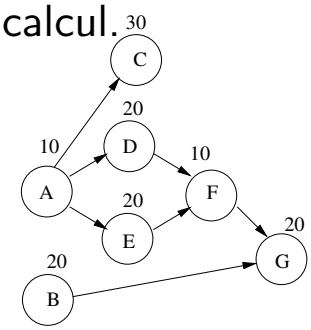
C40link	DMA-I	DMA-O
DMA-I	-	50%
DMA-O	50%	-

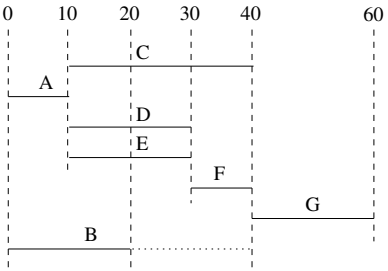
Les DMA en entrée et en sortie sont ralentis de 50% car ils accèdent à une mémoire partagée.

Optimisation de l'implantation distribuée

Chemin critique

Pour calculer le **chemin critique** (CC) d'un graphe d'algorithme étiqueté avec les durées des opérations de calcul et de communication, on associe à chaque opération un segment de longueur égale à sa durée que l'on positionne à sa **date de début au plus tôt** déterminée en fonction des dates de fin de ses prédécesseurs et on détermine sa **date de début au plus tard** en fonction des dates de début de ses successeurs. Un CC correspond à un ensemble de segments sans **marge d'ordonnancement**, c.a.d. tel que chaque segment de cet ensemble ait sa date de début au plus tôt égale à sa date de début au plus tard. S'il y a plusieurs CC on prend leur Max. Pour simplifier on ne considère ci-dessous que les opérations de calcul.





3 CC : {A, D, F, G} = 60, {A, E, F, G} = 60 et {A, C} = 40+20=60, un chemin non critique {B, G} car B a de la marge

Optimisation de l'implantation distribuée

Heuristique minimisant la latence mono-période stricte, non préemptif 1/4

On utilise une **heuristique gloutonne** de distribution et d'ordonnancement rapide car de faible complexité $\mathcal{O}(np)$ avec $n = Card(O)$ et $p = Card(P)$.

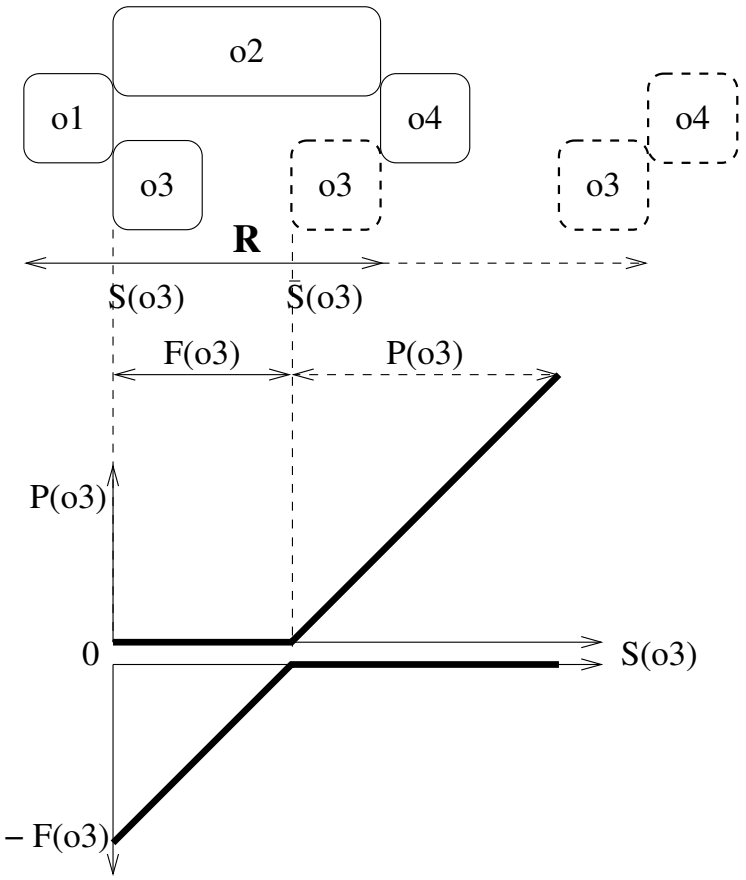
Afin de respecter l'ordre partiel défini par les dépendances entre opérations du graphe d'algorithme, l'heuristique choisit à l'étape i, selon une fonction de coût locale σ, une opération dans le sous-ensemble des opérations, appelé « **candidats** », dont les prédécesseurs ont été distribués et ordonnancés. L'opération ainsi choisie est enlevée de l'ensemble initial d'opérations. La fonction de coût $\sigma(o, p) = P - F$, $o \in O, p \in P$ est appelée **pression d'ordonnancement** (schedule pressure), avec :

- F : **marge d'ordonnancement** (schedule flexibility), différence entre la date d'exécution au plus tôt et la date d'exécution au plus tard ;
- P : **pénalité d'ordonnancement** (schedule penalty), allongement du **chemin critique** dû à la prise en compte des coûts de communication.

A chaque étape i on a un temps d'exécution du graphe d'opérations partiel, appelé « latence partielle ». A la fin de l'heuristique on obtient la latence.

Optimisation de l'implantation distribuée

Heuristique minimisant la latence mono-période stricte, non préemptif 2/4



o1 → (o2, o3) → o4

$S(o3)$ date début au plus tôt o3
$\overline{S}(o3)$ date début au plus tard o3
$F(o3)$ marge de o3
R = CC partiel = latence partielle

Si la durée de la communication due à o1 → o3 quand les opérations sont allouées sur des processeurs différents, est supérieure à la marge d'ordonnancement $F(o3)$ que possède o3, la date de début au plus tôt de o3 est repoussée, ce qui rallonge R. Ainsi la date de début au plus tôt de o4 est repoussée et la latence entrée-sortie de l'algorithme rallongée d'autant.

Optimisation de l'implantation distribuée

Heuristique minimisant la latence mono-période stricte, non préemptif 3/4

$C_i \subseteq O$: **candidats** dont les prédécesseurs ont tous été distribués et ordonnancés.

HEURISTIQUE AAA MONO-PERIODE STRICTE NON PREEMPTIF

1 : $i = 0, V_0 = O$ opérations initiales du graphe d'algorithme

Tant que $V_i \neq \emptyset$ $i = i + 1$

Pour tout $o_j \in C_i \subset V_i$

Pour tout $p_k \in P$ calculer $\sigma(o_j, p_k)$

$$(o_j, p_k) = \min_{(o'_j, p') \in C_i \times P} \sigma(o'_j, p')$$

Fin pour tout

$$(o_j, p) = \max_{(o'_j, p') \in C_i \times P} \{\sigma(o'_j, p')\}$$

mettre sa date de début d'exécution dans la table d'ordonnancement du processeur p ainsi que celles des opérations de communication induites, et leurs durées, calculer la latence partielle, $V_i = V_{i-1} - \{o_j\}$

Fin pour tout

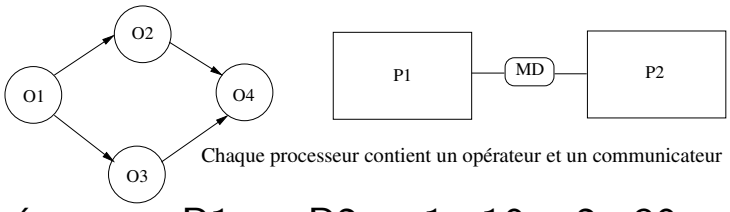
Fin tant que

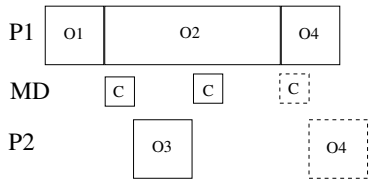
Si latence (dernière latence partielle) $\leq$ contrainte de latence **Fin**

Sinon modifier les contraintes de distribution/ordonnancement ou augmenter le parallélisme potentiel de l'algorithme $O = O'$ **Aller a** 1

Optimisation de l'implantation distribuée

Heuristique minimisant la latence mono-période stricte, non préemptif 4/4





Durées : sur P1 ou P2 : o1=10, o2=30, o3=10, o4=10, sur M : integer=5

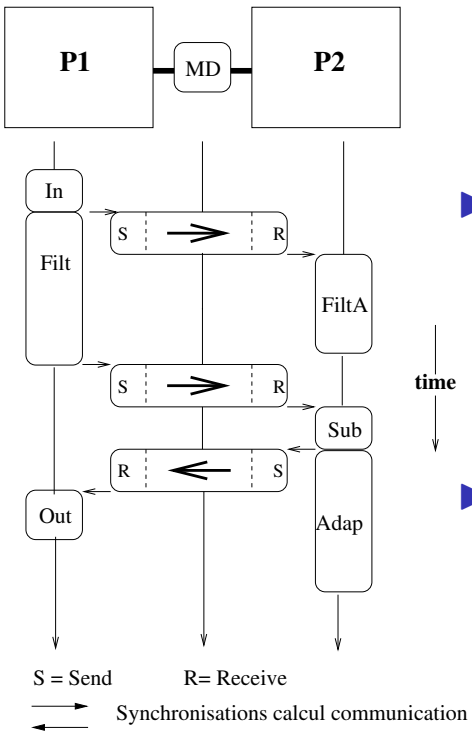
i	(o, p)	$\sigma(o, p)$	min	max	$V_0 = \{o1, o2, o3, o4\}$
1	(o1,P1)	10	(o1,P1)	(o1,P1)	$C_1 = \{o1\}\, V_1 = \{o2, o3, o4\}$
2	(o2,P1)	10+30=40	(o2,P1)	(o2,P1)	$C_2 = \{o2, o3\}$
	(o2,P2)	(10+5)+30=45			
	(o3,P1)	10+10=20	(o3,P1)		
	(o3,P2)	(10+5)+10=25			$V_2 = \{o3, o4\}$
3	(o3,P1)	40+10=50			$C_3 = \{o3\}$
	(o3,P2)	(10+5)+10=25	(o3,P2)	(o3,P2)	$V_3 = \{o4\}$
4	(o4,P1)	(40+**0**)+10=50	(o4,P1)	(o4,P1)	$C_4 = \{o4\}$
	marge o3	**E(o3)+5 < S(o4)**			
	(o4,P2)	(40+5)+10=55			$V_4 = \{\emptyset\}$

Temps d'exécution monoprocesseur 60, CC = 50, latence = 50, accélération sans comm. = 60/50 = 1.2, accélération avec comm. = 1.2

Optimisation de l'implantation distribuée

Résultat de l'adéquation : table d'ordonnancement

Egaliseur implanté sur deux processeurs avec chacun un **opérateur**, un **communicateur** et une **SAM**, communicant par passage de messages via un médium point-à-point.

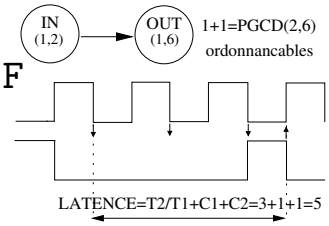


- Le résultat de l'adéquation est un **graphe d'implantation** dont l'ordre partiel est compatible avec l'ordre partiel de l'algorithme. Il est utilisé pour produire la table d'ordonnancement.
- Chaque opération de communication formée d'un envoi S et d'une réception R d'un message de donnée, est exécutée sur les processeurs P1 et P2 par les communicateurs. Le MD est passif. Idem pour l'écriture et la lecture en mémoire partagée.
- Pour chaque processeur et médium du graphe d'architecture la table d'ordonnancement, correspondant au graphe d'implantation, donne les **dates de début** et de **fin** des opérations de calcul et de communication. Longueur relative à durée.

Optimisation de l'implantation distribuée

Heuristique minimisant la latence multi-période stricte, non préemptif

En multi-période les tâches dépendantes doivent avoir des périodes **égales ou multiples** pour ne pas perdre de données.



`HEURISTIQUE AAA MULTI-PERIODE STRICTES NON PREEMPTIF`

1 `Assignation` :
Pour tout $\tau_i(C_i, T_i)$ tâche de WCET C_i et de période T_i
Pour tout P_j processeur
Si T_i multiples $\wedge \sum_{i=1}^{n} C_i^{P_j} \leq PGCD(T_i)$ **Alors** $a_j = ok$ **Sinon** $a_j = nok$
Fin pour tout
Si $\forall j\ a_j = nok$ **Alors** l'heuristique ne trouve pas d'ordonnancement
Fin pour tout

2 `Déroulement` : **Pour tout** τ_i copier la tâche autant de fois que le rapport entre l'hyperpériode (PPCM des périodes T_i) et la période de la tâche τ_i et ajouter les arcs supplémentaires nécessaires

3 `Ordonnancement` : **Pour tout** τ_i tâche assignée et déroulée, calculer sa date de début d'exécution en tenant compte du **coût des communications** **Si** cette tâche à été assignée à plusieurs processeurs **Alors** choisir celui **minimisant** la latence **Si** elle respecte son échéance la mettre avec sa date dans la table d'ordonnancement **Sinon** l'heuristique ne trouve pas d'ordonnancement

Optimisation de l'implantation distribuée

Heuristique minimisant la latence multi-période, préemptif 1/2

Périodes **égales ou multiples** pour ne pas perdre les données de tâches dépendantes.

`HEURISTIQUE AAA MULTI-PERIODE PREEMPTIF`

Pour tout τ_i d'un ensemble de tâches

  Pour tout p_k processeur

    Il faut que dans sa première instance τ_i^1 ne dépasse pas son échéance d_i en tenant compte des communications avec ces prédécesseurs pour qu'elle soit ordonnançable **Si** cette **condition nécessaire d'ordonnançabilité** est vérifiée **Alors** calculer sa marge d'exécution $m_i^k = d_i - (r_i + R_i)$ **Sinon** les τ_i ne sont pas ordonnançables

  Fin pour tout

  Allouer cette tâche τ_i sur le processeur pour lequel elle a le plus grand m_i^k ce qui revient indirectement à minimiser la latence

Fin pour tout

Initialiser les dates d'appel de l'ordonnanceur de chaque processeur

Pour tout processeur

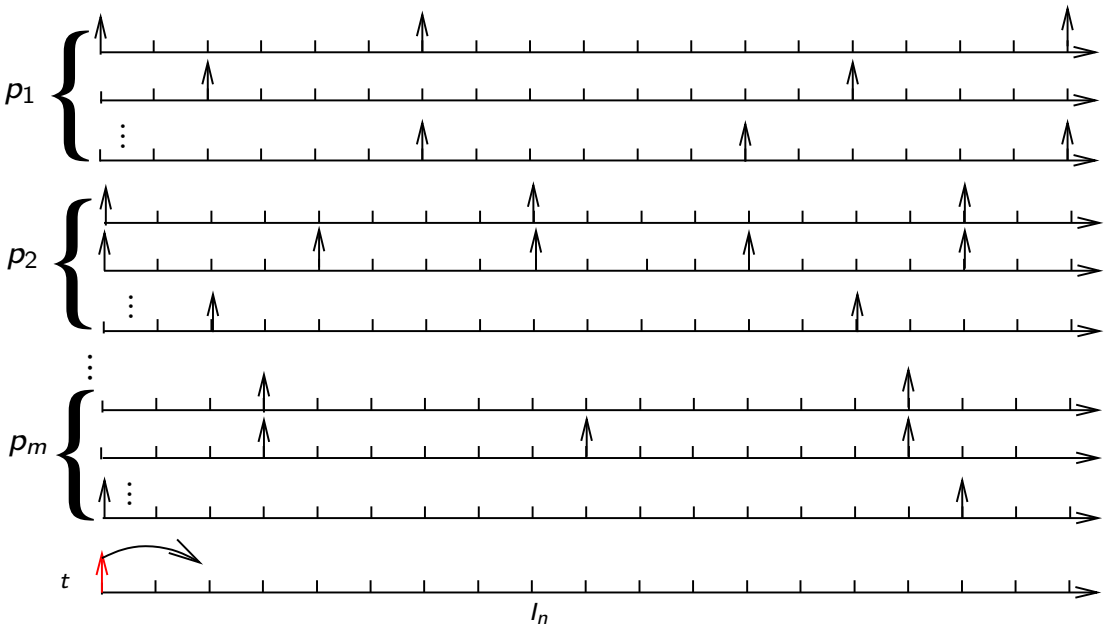
  Si son ordonnanceur est appelé **Alors** déterminer si la tâche allouée est ordonnançable en utilisant la **condition suffisante d'ordonnançabilité monoprocesseur**, calculer le prochain appel de l'ordonnanceur qui peut être **une** date de **consommation de données** lorsque des tâches sont dépendantes

Fin pour tout

Si toutes les tâches τ_i sont ordonnançables **Alors** créer une table d'ordonnancement pour chaque processeur **Sinon** l'heuristique ne trouve pas d'ordonnancement

Optimisation de l'implantation distribuée

Heuristique minimisant la latence multi-période, préemptif 2/2



Pour chaque processeur p_k, vérifier si son ordonnanceur est appelé

Utiliser la condition d'ordonnançabilité monoprocesseur

Déterminer la prochaine date d'appel d'un ordonnanceur la plus proche

Optimisation de l'implantation distribuée

Heuristique minimisant la cadence, accélération

Heuristique pour la cadence

- Recherche des boucles critiques
- Réallocation des mémoires (retiming) associées aux retards
- Conduit à augmenter la latence

Accélération pour une architecture homogène

$$\text{Accélération } A = \frac{\text{somme des durées de toutes les tâches (opérations)}}{\text{durée chemin critique sans communications}}$$

$$\lceil A \rceil = [Card(\mathcal{P})] = p = \begin{cases} \text{Nombre maximal de processeurs } (nbmaxproc) \\ \text{en } \textbf{parallélisme effectif} \text{ nécessaire,} \\ \textbf{en ne tenant pas compte des durées} \\ \text{de communication,} \\ \text{pour exploiter le } \textbf{parallélisme potentiel} \end{cases}$$

nbmaxproc en **parallélisme potentiel** ne tenant pas compte des durées $\geq$
nbmaxproc en **parallélisme effectif** ne tenant pas compte des durées de com. $\geq$
nbmaxproc en **parallélisme effectif** tenant compte des durées

Optimisation de l'implantation distribuée

Heuristique pour minimiser le nombre de processeurs

On peut essayer de minimiser le nombre de processeurs donné au départ.
Pour cela on utilise une **méta-heuristique** (différent de **métaheuristique**) qui appelle une heuristique.

`META-HEURISTIQUE AAA`

$nbProc$ = Nombre initial de processeurs = $[Card(\mathcal{P})] = p$
Exécuter `HEURISTIQUE AAA`
Tant que contrainte de latence satisfaite
    $nbProc = nbProc - 1$
    Exécuter `HEURISTIQUE AAA`
Fin tant que

Génération automatique de code

Généralités 1/2

Les systèmes temps réel critiques étant spécifiés selon une approche **LTT**, on privilégie la **génération de code TT** fondée sur un ordonnanceur hors ligne déclenché par le temps plutôt que **ET** fondée sur un ordonnanceur en ligne déclenché par des interruptions. La génération de code TT repose sur une **analyse d'ordonnançabilité hors ligne** dans les cas **non préemptif et/ou préemptif.**

L'ordonnanceur hors ligne est :

- déclenché (appelé) quand un **compteur de temps discret périodique** (compromis entre précision et appels inutiles de l'ordonnanceur) ou **non périodique** (moins d'appels, contrôle de la dérive du compteur), passe par une certaine valeur. Chaque fois que l'ordonnanceur est appelé, il lit dans la table d'ordonnancement la prochaine opération à exécuter et, de plus dans le cas non périodique, il lit la valeur avec laquelle il initialise le compteur ;
- ou ≪ **auto-déclenché** ≫, sans compteur de temps, sous la forme d'un programme qui boucle sur les opérations de la table d'ordonnancement.

Pour simplifier la gestion du temps **on se place par la suite dans ce dernier cas**. L'exécutif est alors, pour chaque processeur, un code qui séquence les opérations contenues dans la table d'ordonnancement.

Génération automatique de code

Généralités 2/2

- Le **code de l'exécutif** est un code système qui contrôle (ordonnancement, conditionnement, répétition) le **code applicatif** associé aux opérations spécifiées dans l'algorithme (capteur, actionneur, calcul).
- Le code de l'exécutif est **synthétisé sur mesure**. Il tire parti au mieux des caractéristiques de l'application. Le code système a un faible coût, facile à déterminer et constant, ce qui le rend déterministe.
- Le coût du code système doit être pris en compte le plus précisément possible pour que l'analyse d'ordonnançabilité soit sûre.
- Le code de l'exécutif synthétisé sur mesure peut faire appel à un code d'**exécutif résident** qui est générique et ne tire que partiellement parti des caractéristiques de l'application. Le code de l'exécutif résident est en général en ligne et a donc un coût plus important difficile à déterminer et non déterministe, mais a l'avantage d'être ≪ standard ≫.
- On génère automatiquement pour chaque processeur un **pseudo code**, appelé **macro-code**, afin qu'il soit **indépendant de l'architecture**. Il est formé d'un **macro-code de l'exécutif** et d'un **macro-code applicatif**.

Génération automatique de code

Génération du macro-code à partir du résultat de l'adéquation 1/2

Le **macro-code de l'exécutif** est une **boucle infinie** séquençant des :

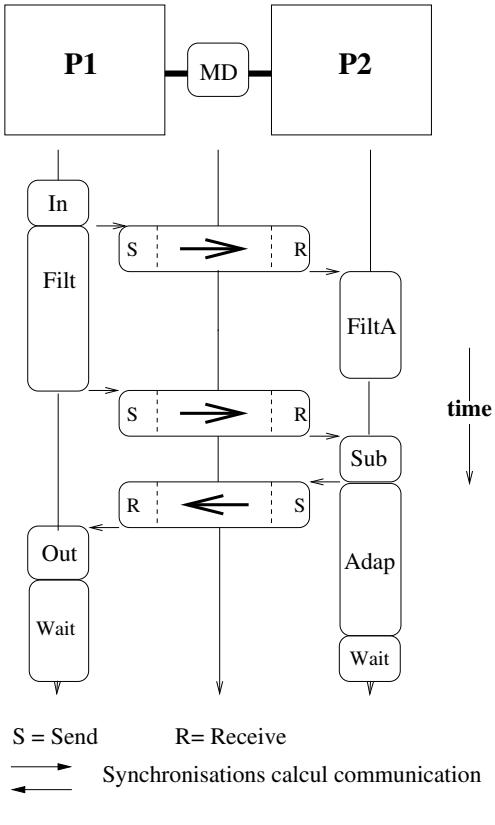
- **macro-instructions système** réalisant :
  - **conditionnement** : choix des opérations en fonction d'une condition ;
  - **répétition** : d'opérations ;
  - **attente (`wait`)** : délai à la fin d'une opération pour assurer sa période ;
  - **communication** inter-processeur : pour les SAM envoi d'un message de donnée d'un communicateur (`send`) et réception de ce message par un communicateur (`recv`), et pour les RAM écriture d'une donnée (`write`) et lecture (`read`) de cette donnée ;
  - **synchronisation** *intra-processeur* et *inter-processeur* ;
- **macro-instructions applicatif** correspondant aux **opérations** distribuées et ordonnancées sur ce processeur, un tampon est associé à chaque sortie d'une opération.

Les synchronisations assurent que même s'il y a des **variations sur les durées des opérations** l'ordre partiel, selon lequel elles s'exécuteront, sera compatible avec l'ordre partiel du graphe d'algorithme initial : conceptions dites ≪ insensibles aux latences ≫ (LID Latency Insensible Design).

Génération automatique de code

Génération du macro-code à partir du résultat de l'adéquation 2/2

Architecture à deux processeurs avec chacun un **opérateur**, un **communicateur** et une **SAM**, communicant par passage de messages via un médium point-à-point.



Opérateur de P1 : macro-boucle d'opérations, macros (`In`, `Filt`, `Out`, `Wait`).
Communicateur de P1 : macro-boucle de communications, macros (`send`, `send` et `recv`).
Opérateur de P2 : macro-boucle d'opérations, macros (`FiltA`, `Sub`, `Adap`, `Wait`).
Communicateur de P2 : macro-boucle de communications, macros (`recv`, `recv`, `send`).
Chaque macro-boucle d'opérations et chaque macro-boucle de communications, contiennent des macros supplémentaires de **synchronisations intra-processeur** et **inter-processeur**.
Les macros **wait** assurent un auto-déclenchement périodique qui peut être supérieur à la latence.

Génération automatique de code

Synchronisations dans le macro-code : principes 1/2

Les **synchronisations intra-processeur** sont de deux types :

- intra-répétition : pour assurer l'exécution en parallèle, **correcte au sens de l'ordre partiel initial**, de la séquence unique de calculs et des séquences de communications à l'intérieur d'une répétition infinie ;
- inter-répétition : pour assurer que les répétitions infinies **se succèdent correctement**. Une répétition infinie doit être terminée avant de passer à la suivante, donc il faut que chaque message de donnée envoyé ait bien été reçu via les SAM impliquées ou que chaque donnée écrite dans une mémoire partagée RAM ait bien été lue.

Les macros de synchronisation effectuent de manière atomique un ≪ **lire-modifier-écrire** ≫ dans un sémaphore et se déclinent comme suit :

- intra-répétition : `Pre_full, Succ_full` : signale tampon plein, attend tampon plein ;
- inter-répétition : `Pre_empty, Succ_empty` : signale tampon vide, attend tampon vide.

Génération automatique de code

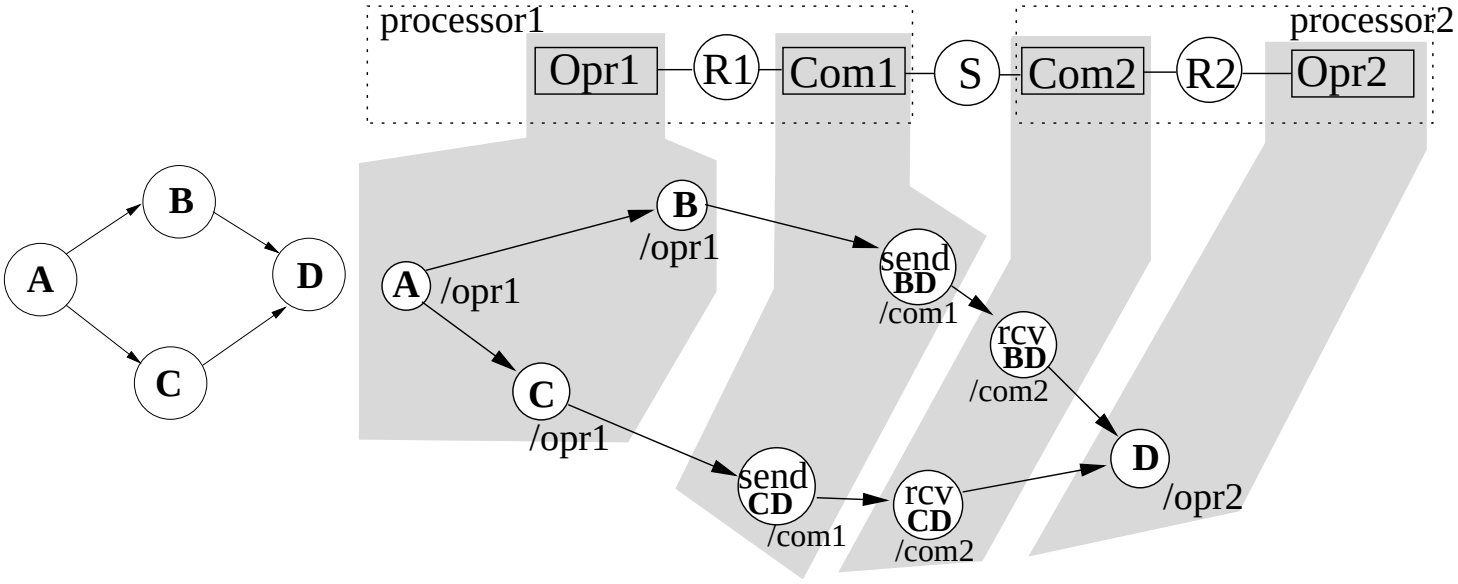
Synchronisations dans le macro-code : principes 2/2

Les **synchronisations inter-processeur** réalisent les synchronisations entre les macro-boucles des communicateurs de plusieurs processeurs interconnectés, à l'aide :

- pour le passage de messages :
  - d'un médium point-à-point : pas de messages de synchronisation car la FIFO est synchronisante ;
  - d'un médium multi-point diffusant : `send` de la donnée à tous les processeurs, reçue à l'aide d'un `sync` par celui qui ne l'utilise pas ;
  - médium multi-point non diffusant : envoi de messages de synchronisation `send_synchro` et reception de messages de synchronisation `recv_synchro` pour les processeurs concernés ;
- pour la mémoire partagée d'un sémaphore supplémentaire :
  - `PreR_full, SuccR_full` : signale mémoire pleine, attend mémoire pleine ;
  - `PreR_empty, SuccR_empty` : signale mémoire vide, attend mémoire vide.

Génération automatique de code

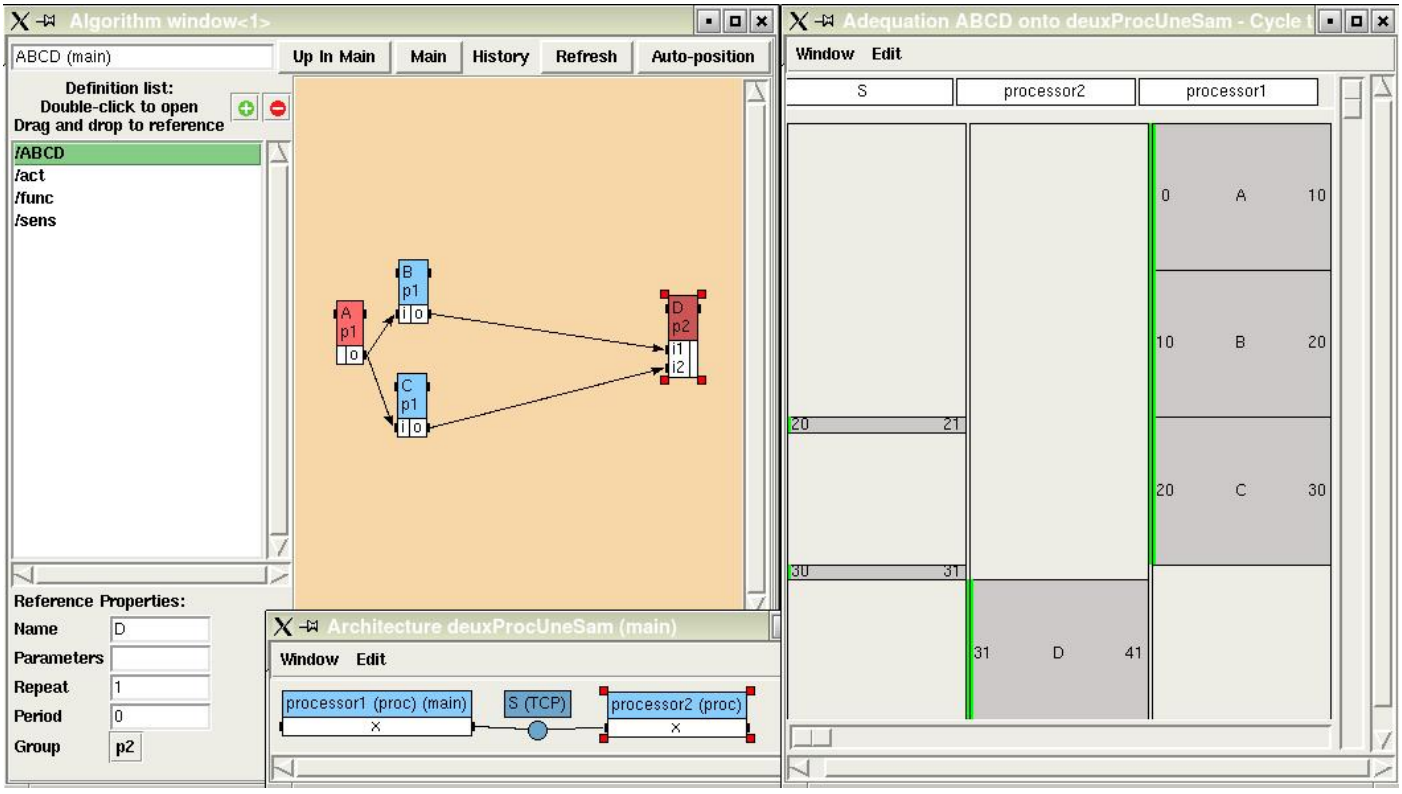
Synchronisations : Graphes d'algorithme ABCD et d'architecture, table d'ordonnancement



Les mémoires R1 et R2 contiennent les tampons mémoires dans lesquels les opérations B et C (resp. D) produisent (resp. consomment) leur donnée.

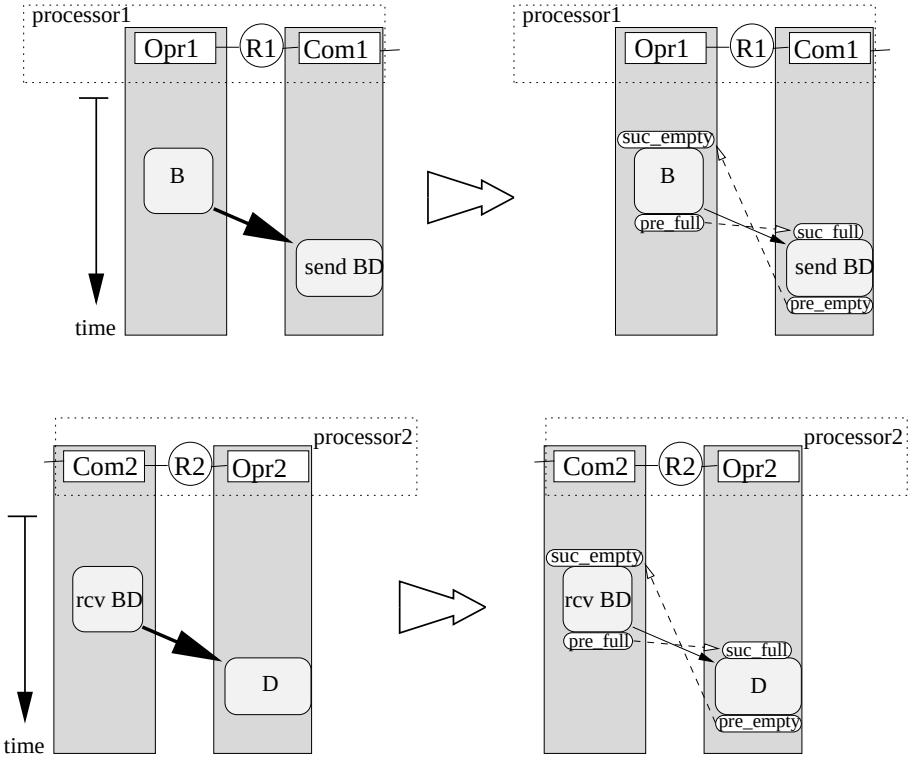
Génération automatique de code

Synchronisations intra-processeur : adéquation, communication par passage de messages point-à-point, algorithme ABCD



Génération automatique de code

Synchronisations intra-processeur : communication par passage de messages point-à-point, `B-Send` et `Receive-D`



Synchronisations intra-répétition (Pre_full, Suc_full) : tampon plein avant exécution `send`.

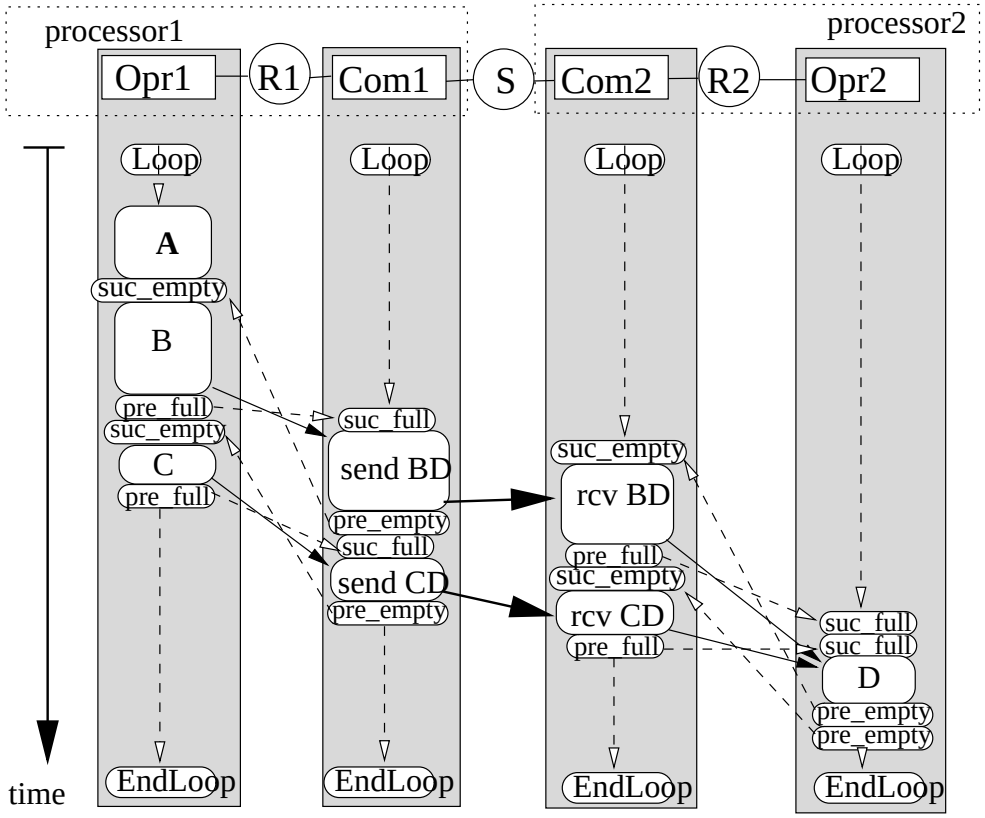
Synchronisations inter-répétition (Suc_empty, Pre_empty) : tampon vide avant envoi donnée.

Synchronisations intra-répétition (Pre_full, Suc_full) : tampon plein avant exécution `recv`.

Synchronisations inter-répétition (Suc_empty, Pre_empty) : tampon vide avant réception donnée.

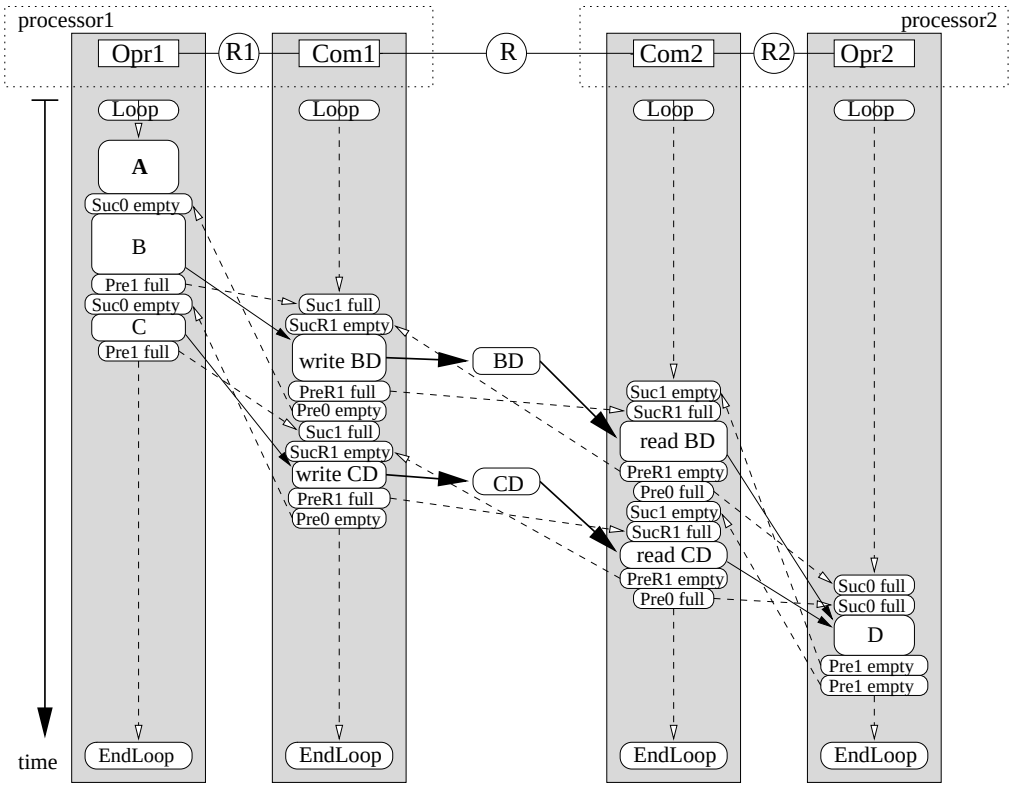
Génération automatique de code

Synchronisations intra-processeur : communication par passage de messages point-à-point, algorithme ABCD



Génération automatique de code

Synchronisations intra-processeur : communication par mémoire partagée, algorithme ABCD



Génération automatique de code

Génération de macro-code : synchronisation intra-processeur communication par passage de messages point-à-point, algorithme ABCD, processor1

```
include(syndex.m4x)dnl
dnl
processor_(proc,processor1,ABCD,
SynDEx-7.0.5 (C) INRIA 2001-2009, 2010-12-20 16:32:09)

semaphores_(
  Semaphore_Thread_x,
  _ABCD_C_o_processor1_x_empty,
  _ABCD_C_o_processor1_x_full,
  _ABCD_B_o_processor1_x_empty,
  _ABCD_B_o_processor1_x_full)

alloc_(int,_ABCD_A_o,1)
alloc_(int,_ABCD_B_o,1)
alloc_(int,_ABCD_C_o,1)
```

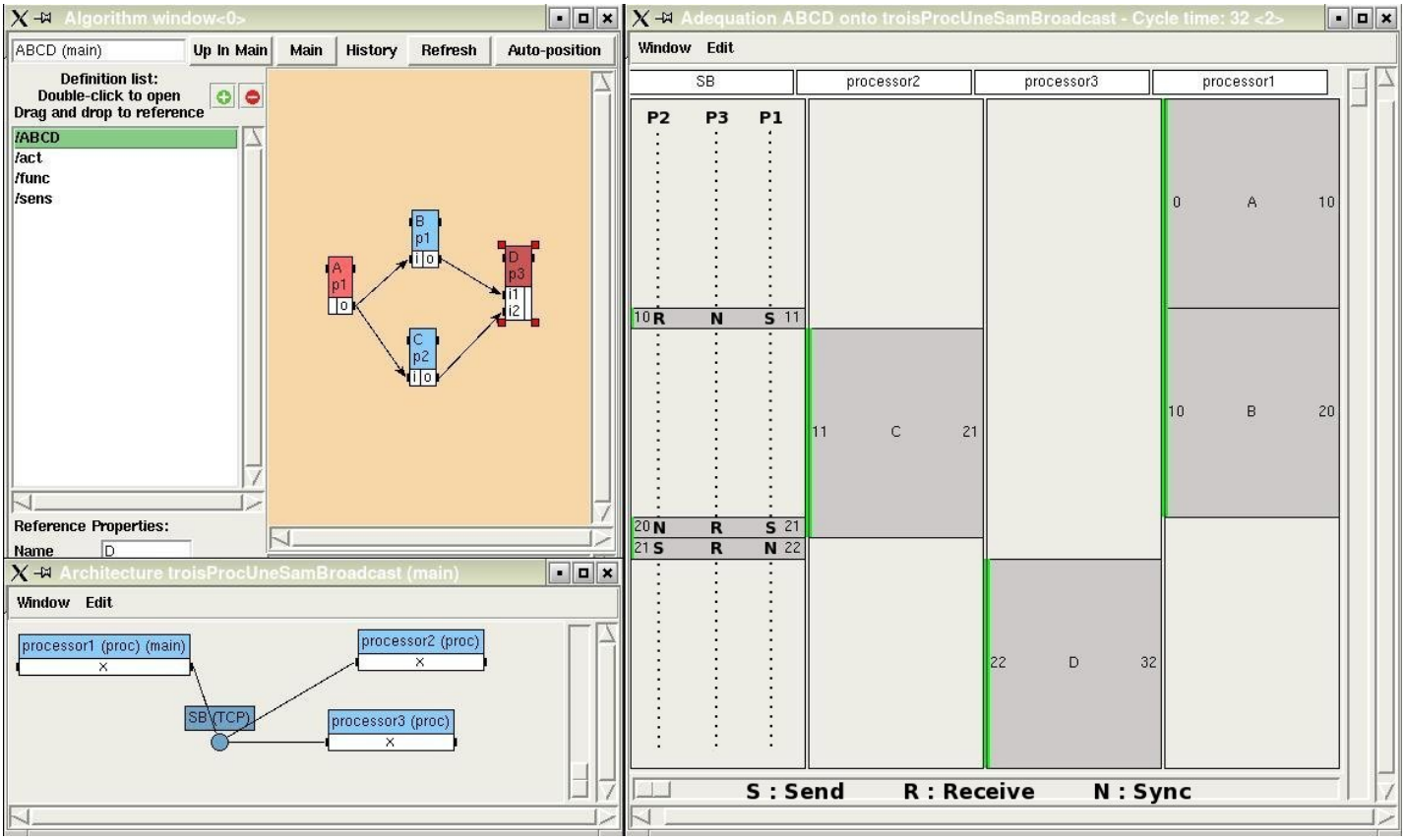
```
main_
  spawn_thread_(x)
  sens(_ABCD_A_o)
  loop_
    sens(_ABCD_A_o)
    Suc0_(_ABCD_B_o_processor1_x_empty,x,_ABCD_B_o,empty)
    func(_ABCD_A_o,_ABCD_B_o)
    Pre1_(_ABCD_B_o_processor1_x_full,x,_ABCD_B_o,full)
    Suc0_(_ABCD_C_o_processor1_x_empty,x,_ABCD_C_o,empty)
    func(_ABCD_A_o,_ABCD_C_o)
    Pre1_(_ABCD_C_o_processor1_x_full,x,_ABCD_C_o,full)
  endloop_
  sens(_ABCD_A_o)
  wait_endthread_(Semaphore_Thread_x)
endmain_

endprocessor_
```

```
thread_(TCP,x,processor1,processor2)
  loadDnto_(,processor2)
  Pre0_(_ABCD_B_o_processor1_x_empty,,_ABCD_B_o,empty)
  Pre0_(_ABCD_C_o_processor1_x_empty,,_ABCD_C_o,empty)
  loop_
    Suc1_(_ABCD_B_o_processor1_x_full,,_ABCD_B_o,full)
    send_(_ABCD_B_o,proc,processor1,processor2)
    Pre0_(_ABCD_B_o_processor1_x_empty,,_ABCD_B_o,empty)
    Suc1_(_ABCD_C_o_processor1_x_full,,_ABCD_C_o,full)
    send_(_ABCD_C_o,proc,processor1,processor2)
    Pre0_(_ABCD_C_o_processor1_x_empty,,_ABCD_C_o,empty)
  endloop_
  saveFrom_(,processor2)
endthread_
```

```
thread_(TCP,x,pr
  loadFrom_(proc

  loop_
    Suc1_(_ABCD_
    recv_(_ABCD_
    Pre0_(_ABCD_
    Suc1_(_ABCD_
    recv_(_ABCD_
    Pre0_(_ABCD_
  endloop_
  saveUpto_(proc
endthread_
```

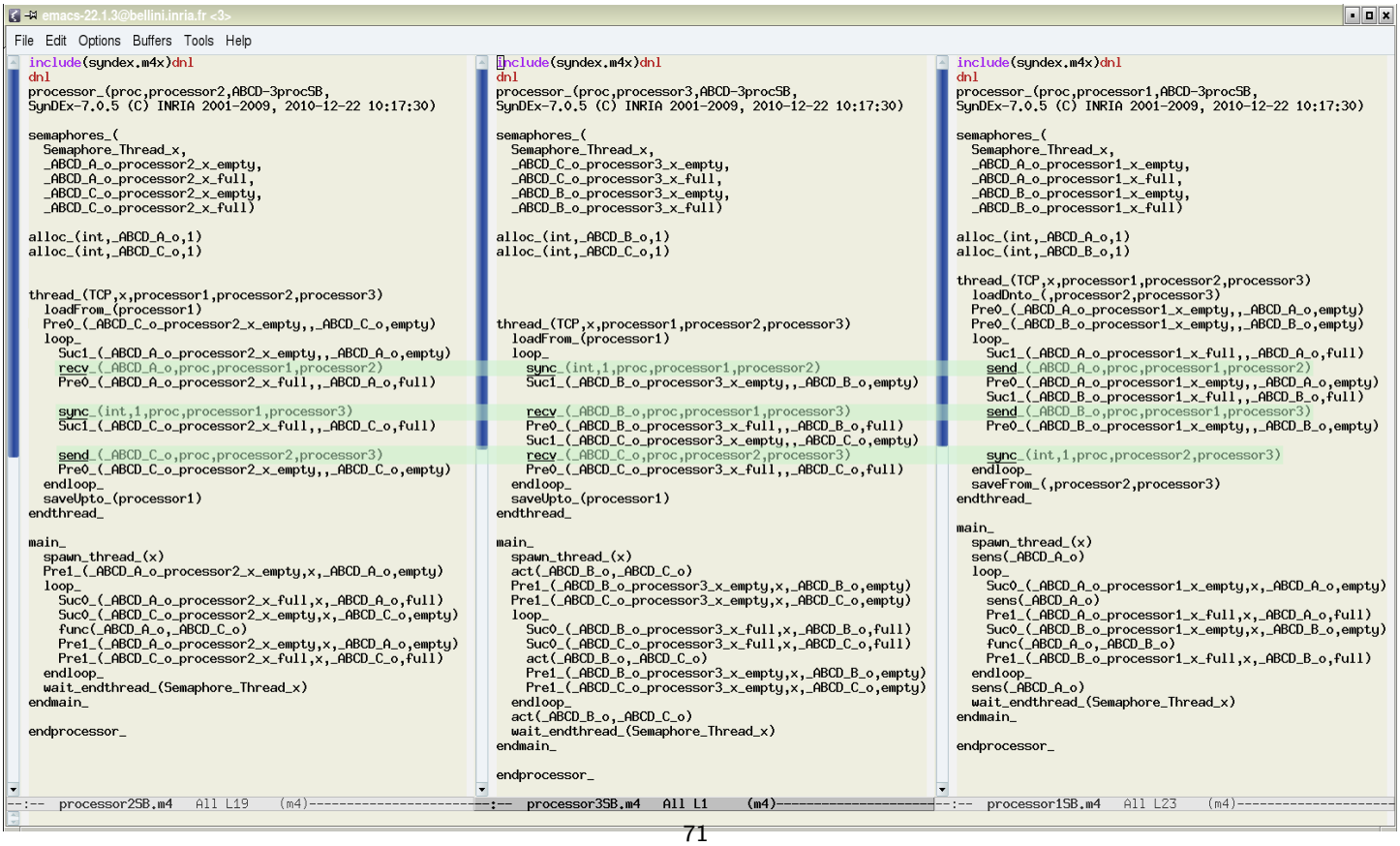
Génération automatique de code

Synchronisations inter-processeur : adéquation, communication par passage de messages multi-point diffusant, algorithme ABCD, 3 processeurs



Génération automatique de code

Génération de macro-code : synchronisations inter-processeur communication par passage de messages multi-point diffusant, algo ABCD, 3 proc



Génération automatique de code

Génération de code exécutable distribué temps réel embarqué 1/2

Les macro-instructions du macro-code de chaque processeur sont traduites en du code source avec le **macro-processeur M4** à l'aide de deux bibliothèques :

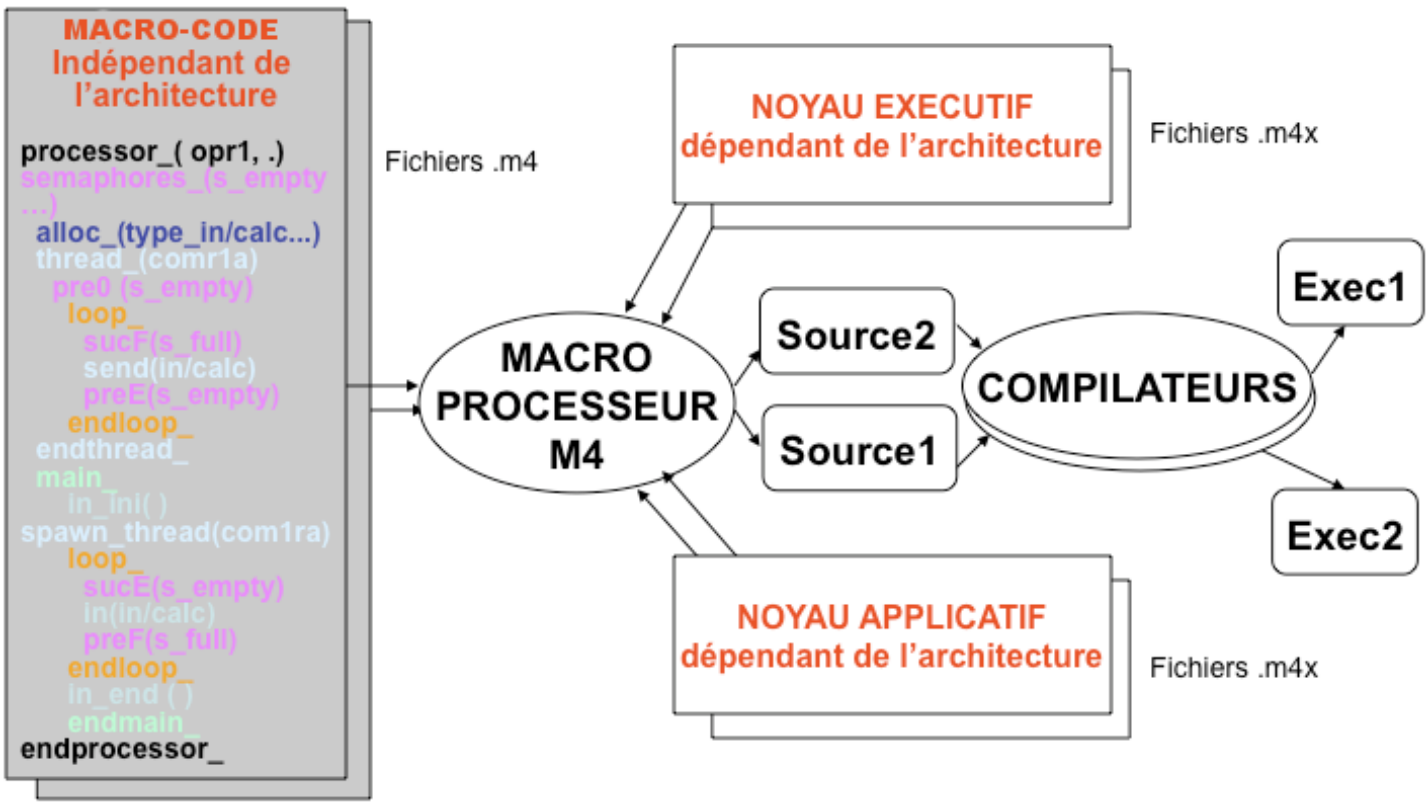
- un **noyau d'exécutif** qui est **dépendant de l'architecture** et de l'éventuel **exécutif résident**, par exemple VxWorks, Osek, Linux/RTAI, Linux/Xenomai, Windows/RTX, etc., décrivant comment chaque macro-instruction système sera remplacée par du code source ;
- un **noyau applicatif** qui est **dépendant de l'architecture** décrivant comment chaque macro-instruction applicatif sera remplacée par du code source.

Après traduction des macro-codes, les codes sources obtenus **sont compilés pour produire des exécutables**. Il faut **déterminer le WCET** C_i de l'exécutable associé à chaque opération de type capteur, actionneur et calcul, afin de pouvoir l'utiliser dans l'analyse d'ordonnançabilité qui produit la table d'ordonnancement.

Dans tous les exécutables obtenus, les synchronisations assurent que l'ordre partiel de l'algorithme est conservé par la génération automatique de code garantissant un fonctionnement en temps réel **sans interblocages**.

Génération automatique de code

Génération de code exécutable distribué temps réel embarqué 2/2



Langage d'implantation et logiciel SynDEx

Fonctionnalités 1/2

Mise en œuvre de la méthodologie AAA

SynDEx est un langage sur lequel sont fondées la spécification fonctionnelle et non fonctionnelle d'un logiciel graphique interactif, lui aussi nommé **SynDEx**, d'aide à l'implantation d'applications de traitement du signal et des images et de contrôle/commande devant s'exécuter en temps réel sur des architectures multicomposant. Il a les fonctionnalités suivantes :

- spécification fonctionnelle :
  - spécification du graphe d'algorithme selon le langage SynDEx ;
  - interface avec des langages spécifiques à un domaine (DSL) : langages Synchrones (ESTEREL/SYNCCHARTS, SIGNAL) (vérifications formelles et simulation), SCICOS (modélisation/simulation de systèmes hybrides), UML2/MARTE (standard OMG profile UML pour le temps réel embarqué), etc., générant un code source en langage SynDEx ;
- spécification non fonctionnelle selon le langage SynDEx :
  - spécification du graphe multicomposant ;
  - caratérisations temporelles ;

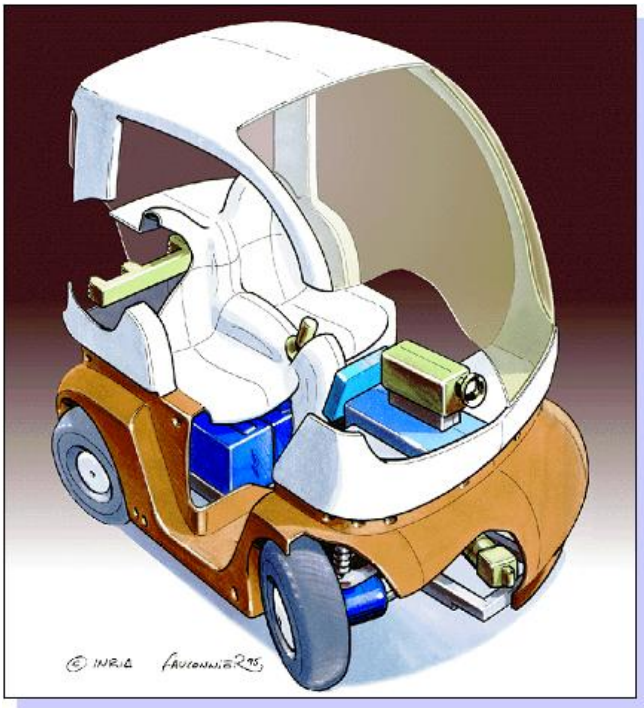
Langage d'implantation et logiciel SynDEx

Fonctionnalités 2/2

- adéquation :
  - analyse d'ordonnançabilité temps réel distribuée produisant une table d'ordonnancement ;
  - optimisations et choix d'une implantation qui préserve les propriétés de la spécification fonctionnelle ;
  - visualisation du diagramme temporel de l'exécution distribuée donnant des mesures de performances simulées ;
- génération automatique d'exécutifs temps réel distribués sans interblocages, synthétisés sur mesure à partir de **noyaux d'exécutif** :
  - pour les processeurs Analog Device ADSP21060, Texas Instrument TMS320C40, Microchip PIC182680, Intel ix86, i8051, i80C196, Motorola MC68332, MPC555, Transputer T80x ;
  - pouvant appeler les exécutifs résidents Linux, Linux/RTAI, Linux/Xenomai, Windows, Windows/RTX pour stations de travail Intel ix86 communiquant par TCP/IP ;
  - mesures de performances temps réel à l'aide de sondes logicielles introduites automatiquement lors de la génération d'exécutifs.

Langage d'implantation et logiciel SynDEx

Exemple CyCab 1/2

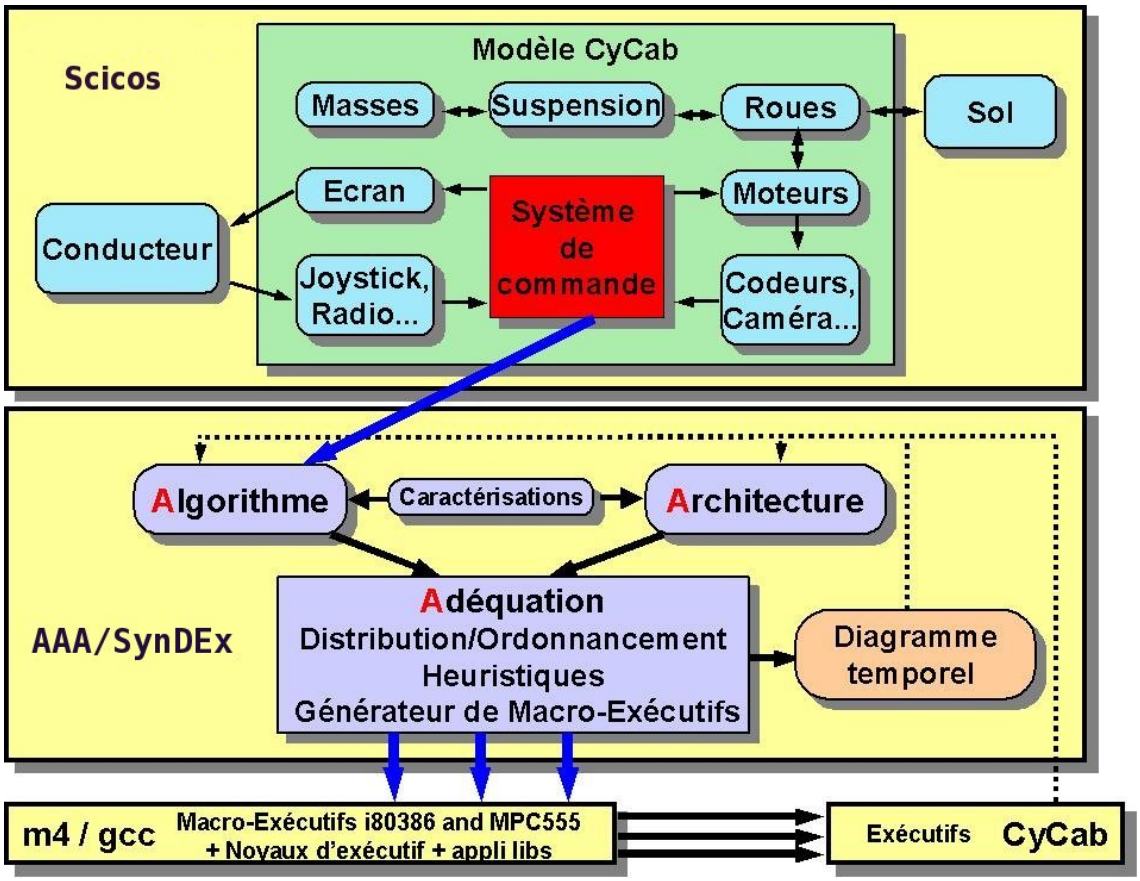


- Vitesse 30km/h
- Moteurs électriques
- 4 roues motrices
- 2 directions AV, AR
- Multi-processeur MPC555 + un PC embarqué
- Bus Can

Industrialisé par Robosoft
www.robosoft.fr

Langage d'implantation et logiciel SynDEx

Exemple CyCab 2/2



Langage d'implantation et logiciel SynDEx

Utilisation

L'utilisateur **spécifie à l'aide de l'IHM** de SynDEx le graphe d'algorithme, le graphe d'architecture, les périodes strictes et les durées d'exécution des opérations, ou il **importe un fichier .sdx** produit par le compilateur de certains langages spécifiques à un domaine (DSL).

L'heuristique d'optimisation AAA effectue l'analyse d'ordonnançabilité et calcule les dates d'exécution de chaque opération de calcul et de communication à partir de leurs période, durée d'exécution d'opérations de calcul et d'opérations de communication. Le générateur de code produit autant de fichiers de macro-code que de processeurs dans l'architecture.

La durée d d'une opération de communication par passage de messages exécutée par les deux communicateurs (`send, recv`) de deux processeurs, est calculée à partir d'un modèle mathématique simple, par exemple : $d = \tau + \delta n$, δ : durée élémentaire de la communication (un élément de donnée), n : nombre d'éléments de donnée (dépendant du type de donnée), τ : temps d'établissement de la communication.

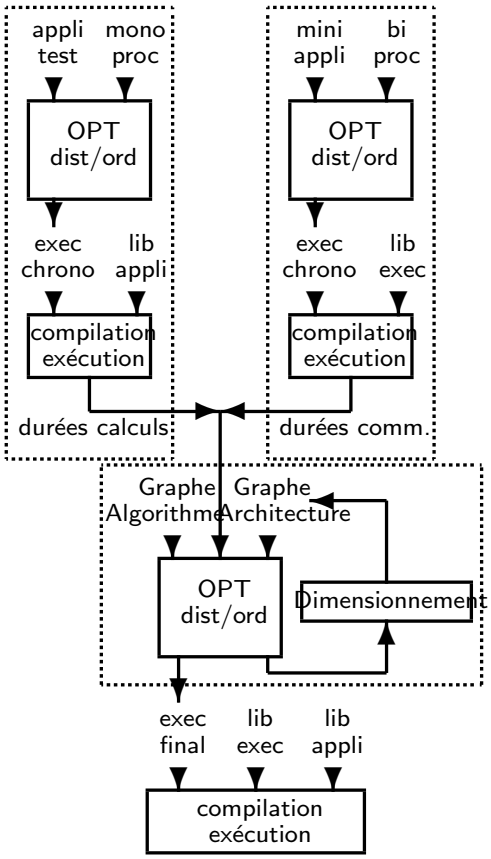
Langage d'implantation et logiciel SynDEx

Mesure des durées d'exécution

On utilise l'option « **Génération de code avec chronomètre** » qui ajoute, pour chaque opération de calcul ou de communication, un premier macro-code chronomètre donnant la date avant l'exécution de l'opération et un second macro-code chronomètre donnant la date après exécution.

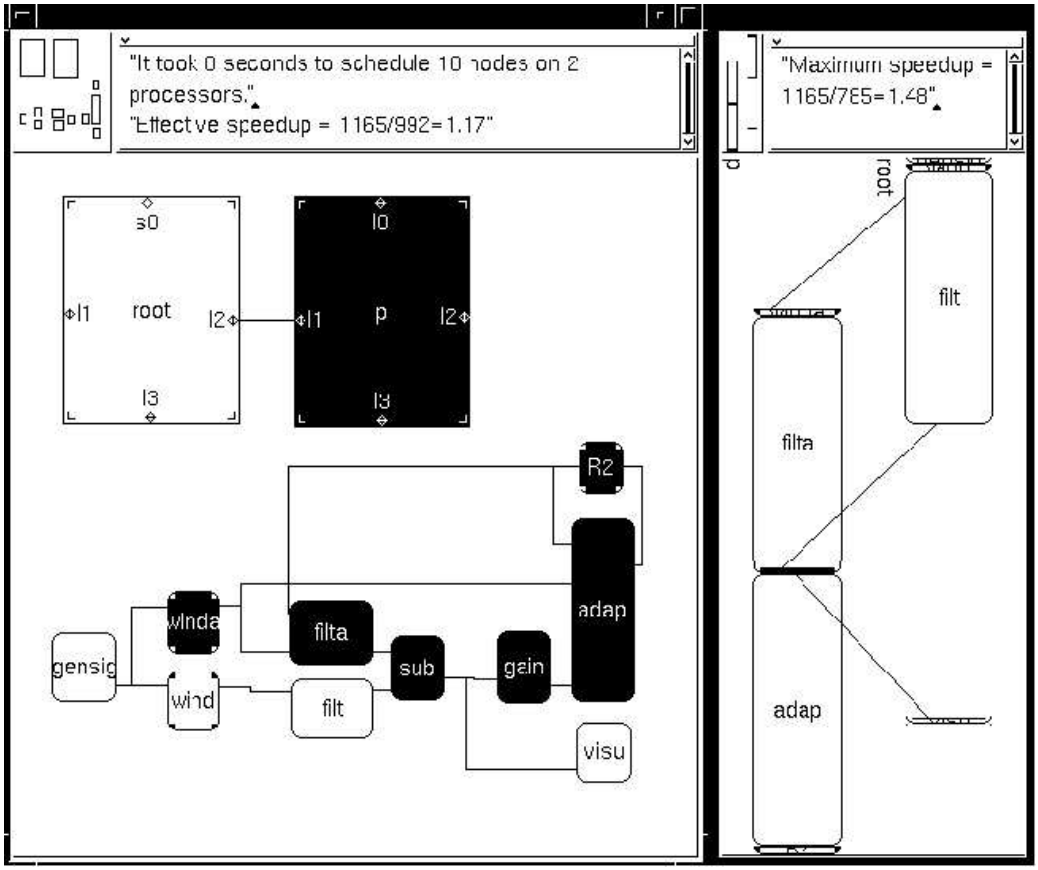
L'application considérée est exécutée en monoprocesseur et la différence entre les dates avant-après donne la durée d'exécution de chacune de ses opérations.

Une application minimale A → B exécutée en biprocesseur, avec chaque médium de communication, donne la durée d'exécution élémentaires de chaque opération de communication.



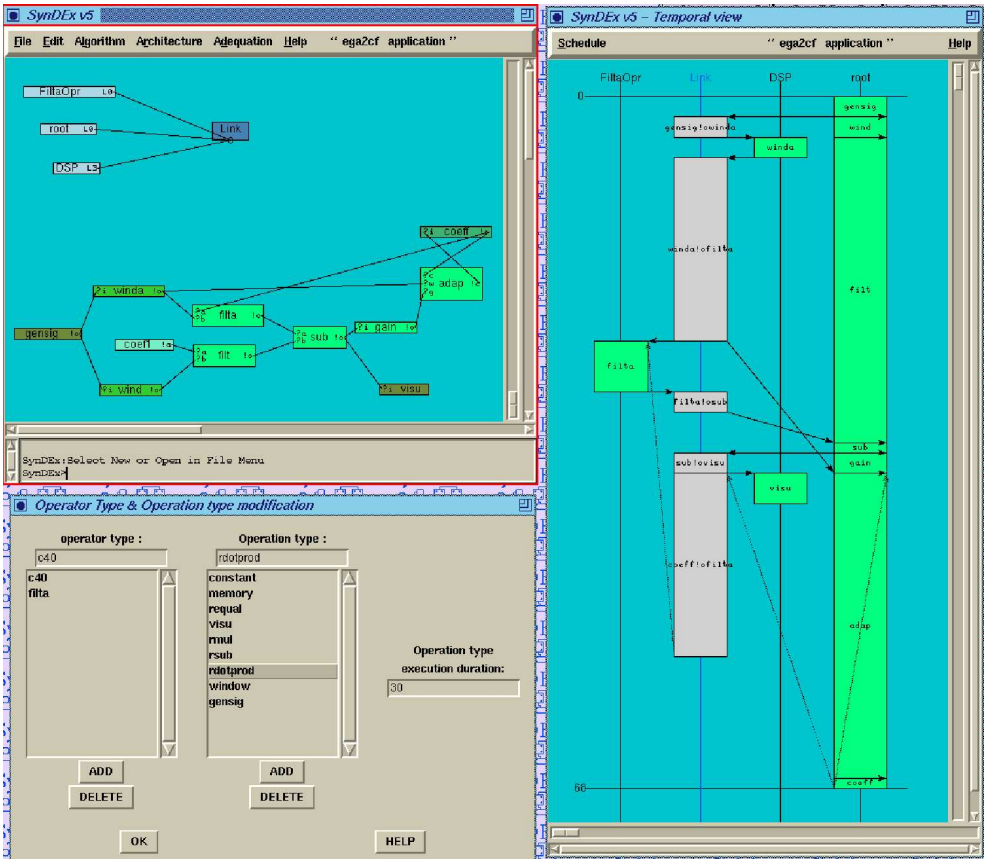
Langage d'implantation et logiciel SynDEx

IHM V4



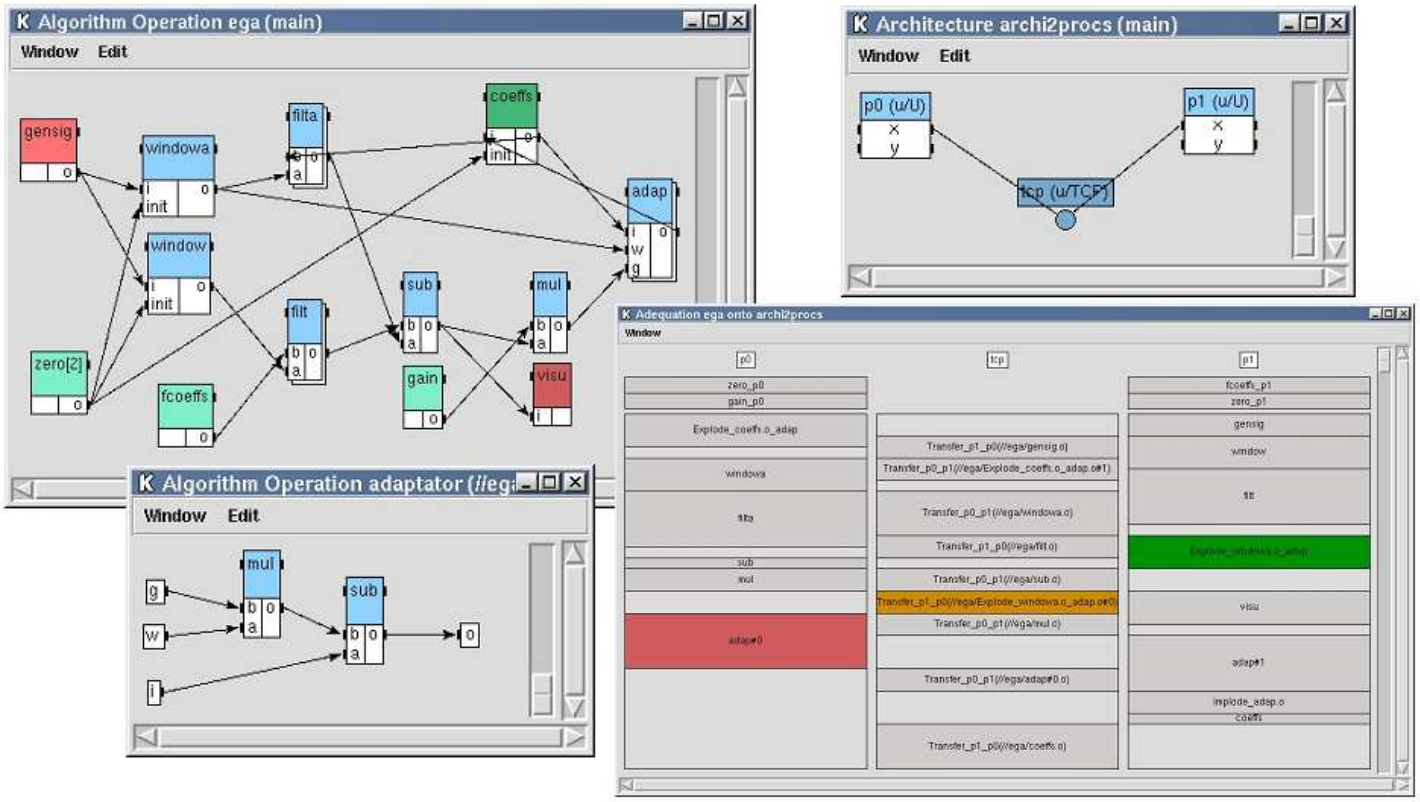
Langage d'implantation et logiciel SynDEx

IHM V5



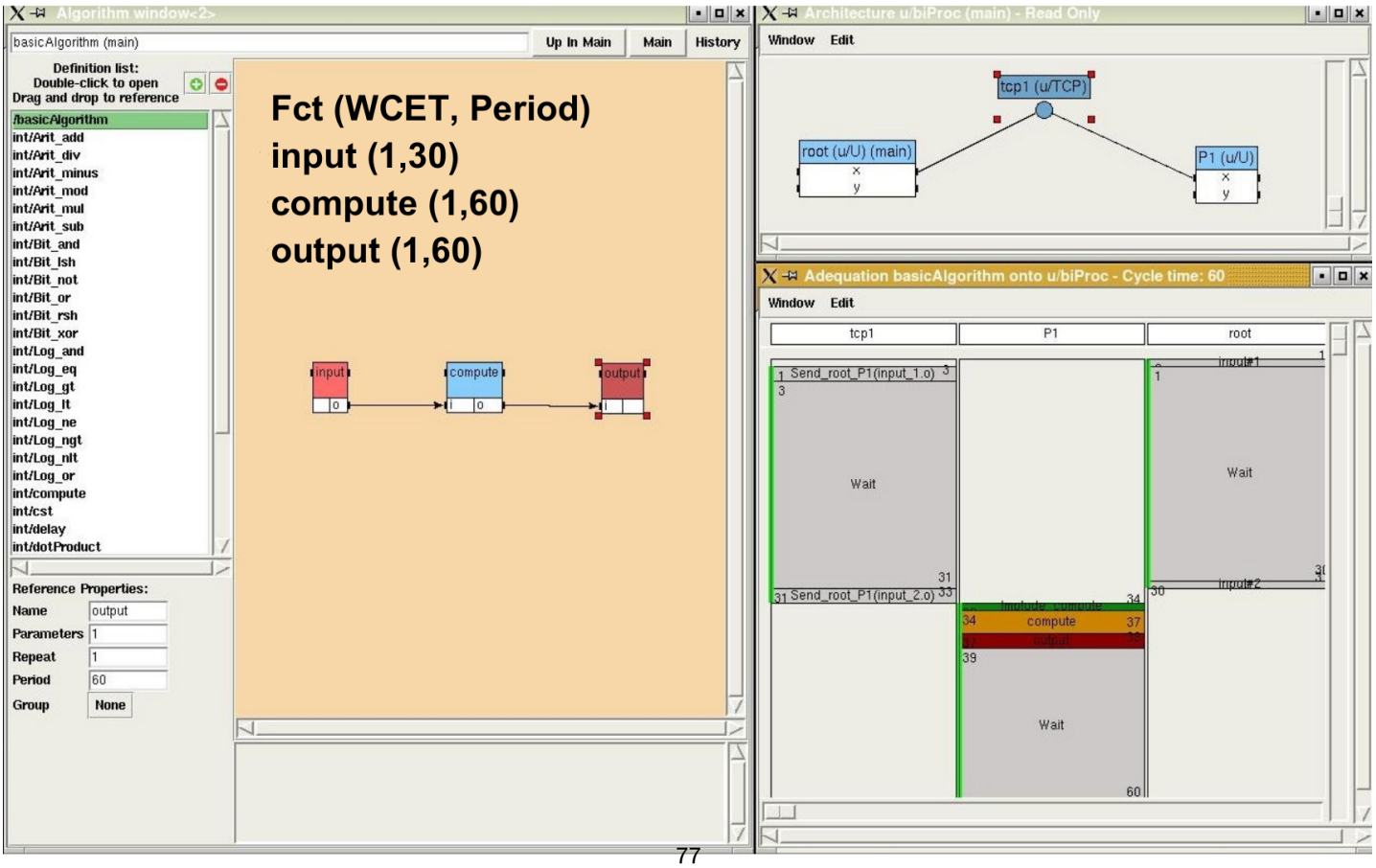
Langage d'implantation et logiciel SynDEx

IHM V6



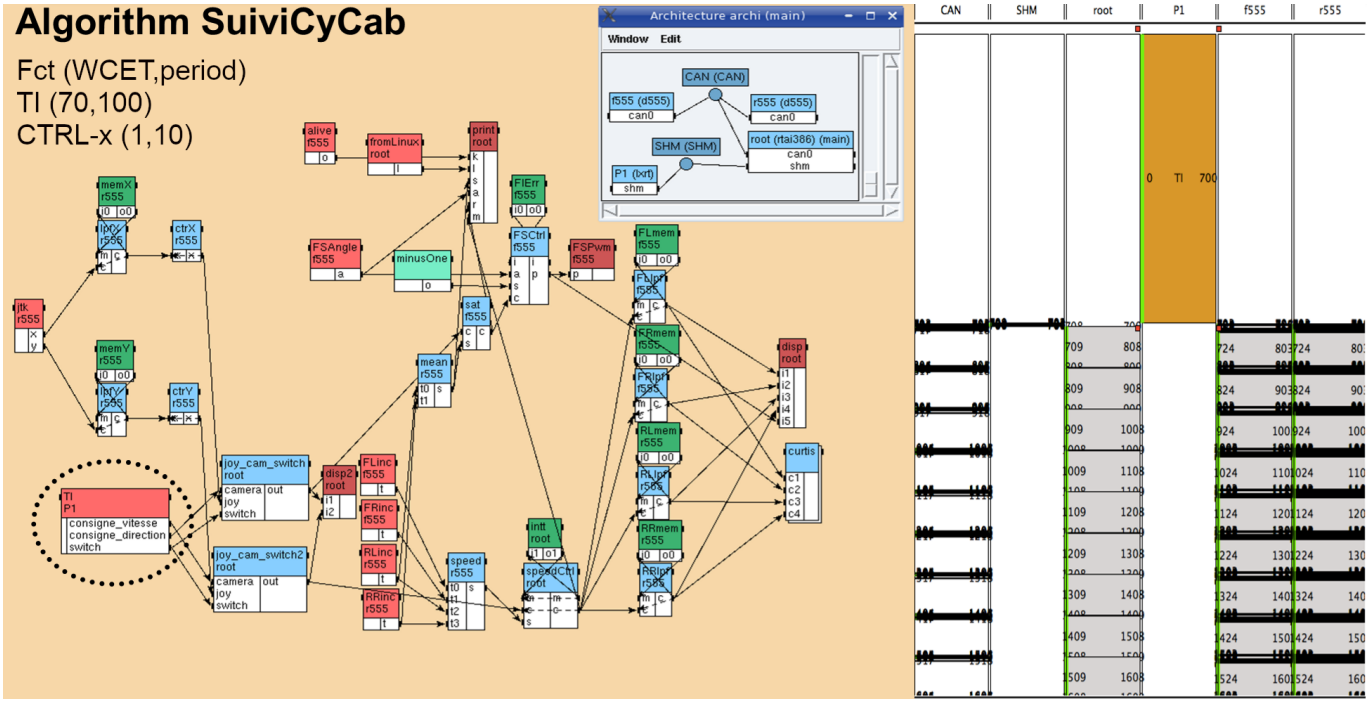
Langage d'implantation et logiciel SynDEx

IHM V7 Algo simple : WCET 1ms, input période 30ms, compute et output périodes 60ms



Langage d'implantation et logiciel SynDEx

IHM V7 Suivi automatique CyCab : TI WCET 70ms période 100ms, CTRL-x WCET 1ms période 10ms



Conclusion

Conclusion

- Pour réaliser une implantation optimisée il faut **maîtriser le lien entre automatique et informatique**.
- Les systèmes temps réel embarqués doivent être **réactifs**, **respecter des contraintes temporelles** et **minimiser des ressources**.
- La spécification fonctionnelle avec certains langages spécifiques à un domaine (DSL) permettent de faire des **vérifications formelles**.
- Le DSL SIGNAL vérifie que l'ordre des événements des sorties est cohérent avec l'ordre des événements des entrées.
- La spécification non fonctionnelle permet de décrire les **ressources matérielles** et les **caractéristiques temporelles**.
- La méthodologie AAA permet de faire des spécifications fonctionnelles et non fonctionnelles, de formaliser des implantations en termes de transformations de graphes, d'étudier les implantations valides en terme **d'ordonnançabilité**, de **minimiser** des critères temporels et de ressources, et de **générer** des exécutifs temps réel embarqués **sûrs**.
- Le langage d'implantation SynDEx **concrétise** la méthodologie AAA.

Conclusion

Sûreté de fonctionnement par construction

Implantation optimisée : adéquation

Temps de développement réduit