Cours

Systèmes distribués temps réel sûrs et optimisés

Méthodologie AAA : adéquation algorithme-architecture

Yves Sorel

INRIA Paris-Rocquencourt
Domaine de Voluceau BP105
78153 Le Chesnay CEDEX
Tél. : 01 39 63 52 60, email : yves.sorel@inria.fr
http://www.syndex.org
http://www.syndex.org/publications

Plan I

Plan II

Contexte et objectifs

Exemples de systèmes embarqués

Automobile

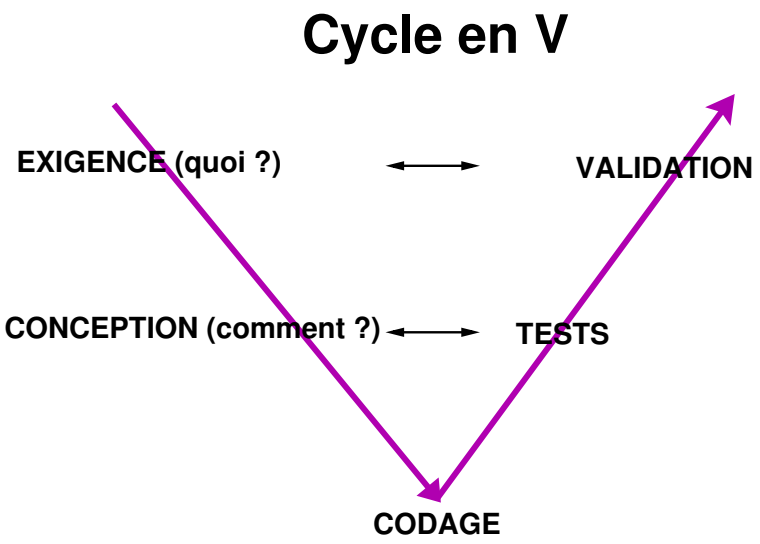
Avionique

Robotique mobile

Telecommunication

Approche système

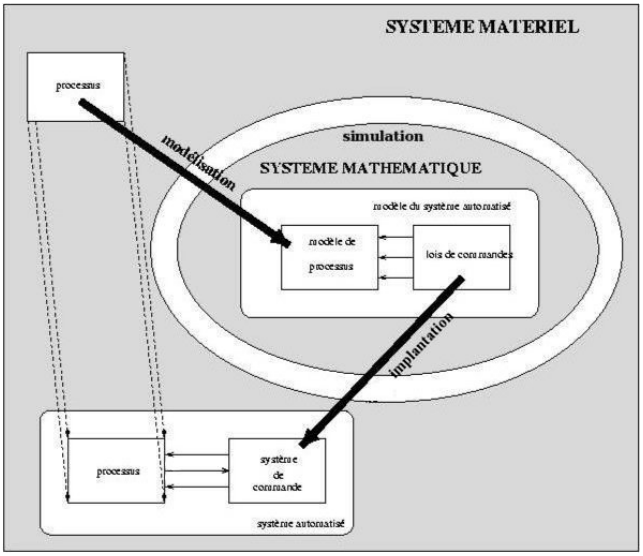
Cycle de développement des systèmes embarqués



On traite la partie basse du cycle en **V** : **Conception** $\leftrightarrow$ **Tests** et **Codage** avec comme but d'automatiser le codage, les tests, ou au moins de les minimiser au maximum. On vise un cycle en **I** offrant une **conception sûre par construction** sans remontée dans le cycle.

Approche système

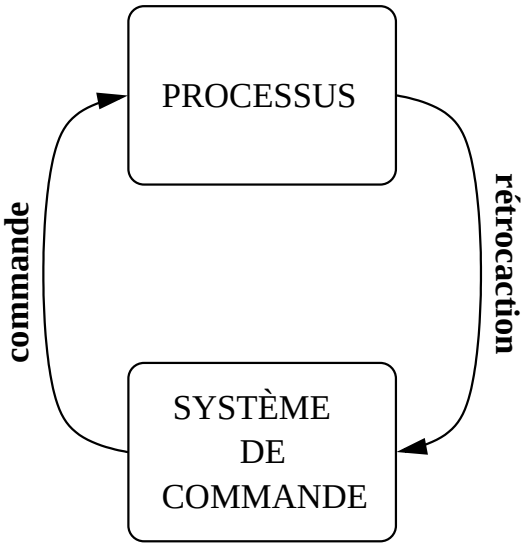
Liens entre AUTOMATIQUE et INFORMATIQUE



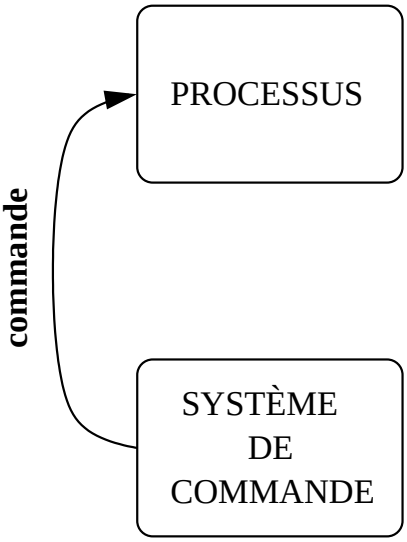
AUTOMATIQUE - modélisation/simulation - **Spécification**
Modélisation/simulation hybride du système : processus domaine continu et/ou discret, système de commande domaine discret.
INFORMATIQUE - implantation - **Conception Codage**
Implantation du système de commande sur processeurs et/ou circuits intégrés spécifiques (CI), puis connexion avec le processus.

Approche système

Structure générale d'un système automatisé



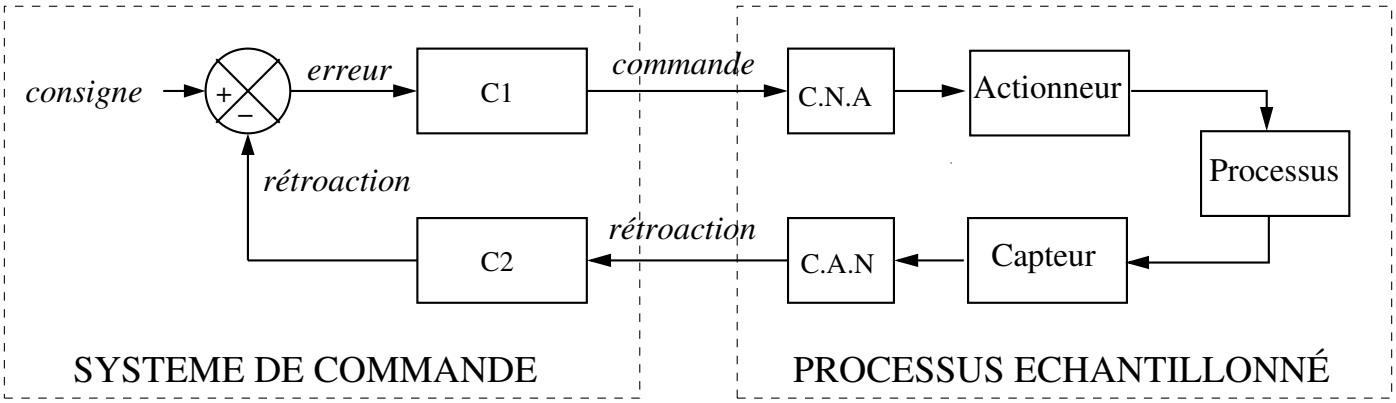
(a) boucle fermée



(b) boucle ouverte

Approche système

Système de commande bouclé sur processus échantillonné



Définitions

Système réactif

Evénements entrée Stimulus	⇒	SYSTEME DE COMMANDE REACTIF ET TEMPS REEL	⇒	Evénements sortie Réaction

Système réactif (Harel, Pnueli 1985) : le système de commande quand il consomme un **événement d'entrée**, exécute des fonctions et **doit** produire un **événement de sortie**.

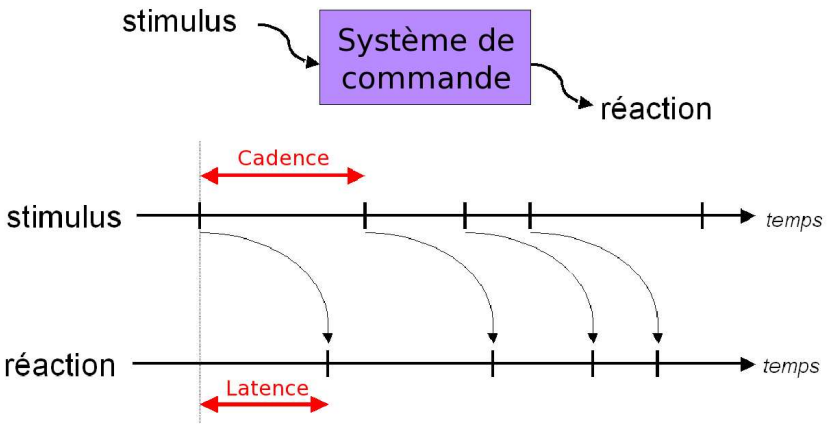
Le système de commande consomme une infinité d'événements d'entrée, valeurs numériques produites par le processus via un **capteur** associé à convertisseur analogique numérique (CAN). Le système de commande produit une infinité d'événements de sortie, valeurs numériques consommées par le processus via un **actionneur** associé à un convertisseur numérique analogique (CNA). Chaque suite infinie d'événements d'entrée ou de sortie, s'appelle un **signal**.

Définitions

Système temps réel

Système temps réel : système réactif devant respecter des contraintes :

- de **cadence** : contrainte sur la durée s'écoulant entre deux événements d'un même signal (périodique, sporadique avec période minimale, apériodique sans période),
- de **latence entrée-sortie** : contrainte sur la durée de la réaction déclenchée par un événement d'un signal d'entrée et produisant un événement d'un signal de sortie.



Définitions

Système temps réel distribué embarqué, déclenché par événements ou temps

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

Système temps réel souple, mou, QoS : on accepte un certain pourcentage de non respect des contraintes, une qualité de service.

Système distribué, parallèle, multiprocesseur, multicœur : pour performances, modularité, rapprocher les calculs des capteurs/actionneurs.

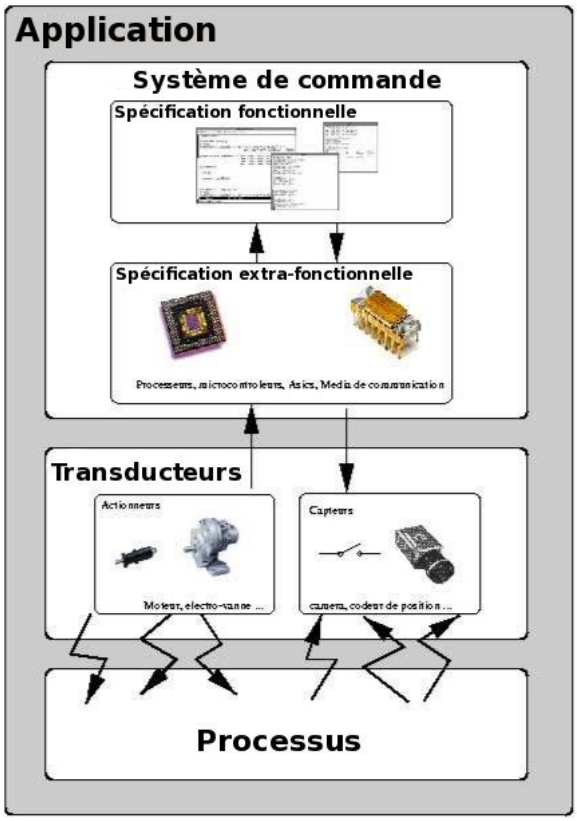
Système embarqué : demande une minimisation des ressources (encombrement, poids, consommation, coût, etc.).

Système déclenché par les événements (ET Event Triggered) : l'état du processus est connu par interruptions des capteurs, actionneurs doivent suivre, flexible, prévision probabiliste, adapté au temps réel souple.

Système déclenché par le temps (TT Time Triggered) : l'état du processus est connu par interrogations périodiques des capteurs relativement à un temps discret (quantum de temps), actionneurs synchronisés, rigide, prévision déterministe, adapté au temps réel dur.

Définitions

Application temps réel



Domaines d'application

Systèmes grand public et à large échelle

- **Systèmes grand public**
  - télécoms : téléphone mobile intelligent, modems, etc.
  - électronique automobile : contrôle moteur, aide à la conduite, etc.
  - robotique : véhicule automatique, nettoyeur, robot industrie, etc.
  - électronique médicale : implants, suivi patient à distance, etc.
  - domotique : télésurveillance, automatisation tâches ménagères, etc.
  - équipements audio et vidéo : baladeur, set-up box, télévision HD, etc.

On vise la minimisation du coût

- **Systèmes à large échelle**
  - aéronautique et spatial
  - contrôle du trafic aérien
  - ferroviaire
  - commande de processus industriels
  - centraux télécoms
  - systèmes d'armes

On vise la rapidité du développement

Spécification fonctionnelle

Traitement du signal et des images - contrôle/commande

Spécification des fonctions du système et des relations de **dépendances de données** liant les sorties des fonctions **productrice**s de données au entrées des fonctions **consommatrices** de données.

Automatique

Traitement du signal et des images	$\Rightarrow$	Calculs nombreux Régulier - `For i=1 to N Do`
Contrôle/commande	$\Rightarrow$	Calculs peu nombreux Non régulier - `If cond Then Else` Changement de modes

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

Spécification extra-fonctionnelle

Architecture matérielle

Spécification des composants de l'architecture matérielle et de la manière dont ils sont interconnectés. Spécification de contraintes de distribution et d'ordonnancement des fonctions sur les composants et des dépendances de données sur les média de communication.

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

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

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

Spécification extra-fonctionnelle

Caractéristiques temporelles

Spécification des caractéristiques temporelles associées à chaque fonction de la spécification fonctionnelle. Elles sont de deux types :

- **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,
- **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 données WCCT (Worst Case Communication Time) sur les média de communication.

Spécification de propriétés de **sûreté de fonctionnement** et de **sécurité non traitées dans le cours**.

Implantation optimisée : méthodologie AAA

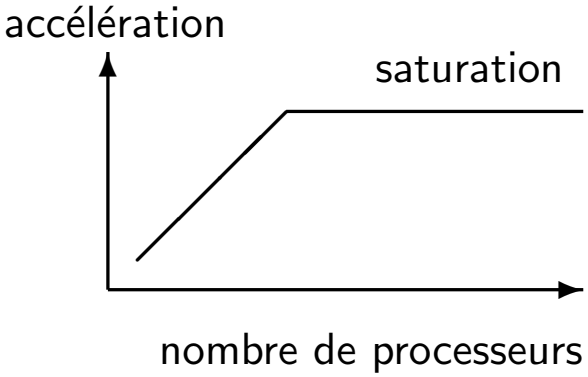
Parallélisme potentiel, parallélisme effectif

Le **parallélisme potentiel** (concurrence) de la spécification fonctionnelle est défini à partir de l'ensemble des fonctions qui ne sont pas dépendantes car, dans ce cas, elles pourront s'exécuter potentiellement en parallèle.

L'**implantation** consiste à choisir sur quel processeur de l'architecture multiprocesseur chaque fonction sera distribuée et ordonnancée.

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

$$\text{Accélération} = \frac{T_{monoprocesseur}}{T_{multiprocesseur}}$$



Implantation optimisée : méthodologie AAA

Rôle du système d'exploitation

Algorithme

Système d'exploitation

Architecture

Algorithme : définition informelle (Al-Khwarizmi, astronome Perse 825) description d'une fonction à l'aide un nombre fini d'instructions choisies dans un ensemble fini d'instructions, plus générique que programme qui suppose le choix d'un langage, définition formelle (Turing, Post 1930).

Système d'exploitation (OS Operating System) : fournit aux fonctions des services permettant d'exploiter l'architecture : programmes, périphériques, mémoire, communications, synchronisations, dépendant éventuellement du temps RTOS (Real-Time OS) appelé aussi **exécutif temps réel** ou **exécutif** par la suite.

Architecture : électronique numérique constitué de processeurs et/ou CI, tous interconnectés.

Implantation optimisée : méthodologie AAA

OS, RTOS, exécutif : distribué réactif temps réel

Fonctionnalités de tous les OS

OS	=	Allocation des ressources matérielles de calcul, mémoire, communications à l'algorithme
+		
Distribué	→	Plusieurs ressources différentes pour chaque type
+		

Fonctionnalités des RTOS

Réactif	→	Ordre réactions = ordre stimuli indépendamment du temps de réaction
+		
Temps réel	→	Allocation conditionnée par l'écoulement du temps physique

Implantation optimisée : méthodologie AAA

Exécutif temps réel : allocation de ressources

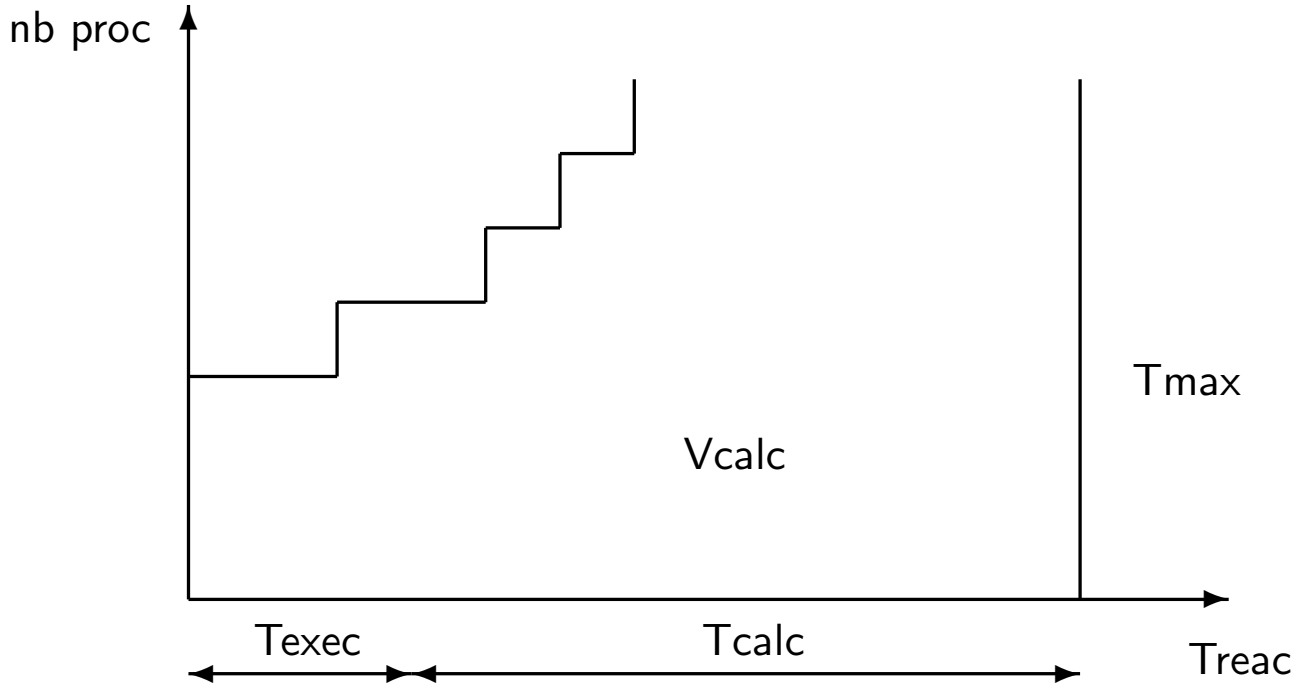
L'exécutif temps réel alloue des ressources

- Distribution : allocation spatiale
- Ordonnancement : allocation temporelle
- Hors-ligne : optimisations et décisions effectuées **avant** l'exécution (connaissance des durées d'exécution)
- En-ligne : optimisations et décisions effectuées **pendant** l'exécution (utilisation de l'horloge temps réel)

Implantation optimisée : méthodologie AAA

Exécutif temps réel : surcoût de l'exécutif

L'exécutif temps réel introduit un surcoût qui croît avec le nombre de processeurs



Implantation optimisée : méthodologie AAA

Objectifs

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

Pour atteindre cet objectif l'exploration s'effectue à partir de **modèles formels** représentant l'algorithme et l'architecture (graphes, ordre partiel, machine à états finie) en effectuant 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 de base

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 $(s_1\, A\, s_2) \Leftrightarrow (s_1$ "est relié à" $s_2)$ par une **arête**. Ce graphe est **orienté** si chaque couple (arête) est ordonné $(s_1, s_2) \neq (s_2, s_1)$, on dit s_1 "précède" s_2. Un couple ordonné est appelé **arc**. Ce graphe est **acyclique** s'il ne possède pas de suite d'arcs $(s_1, s_2) \ldots (s_n, s_1)$.

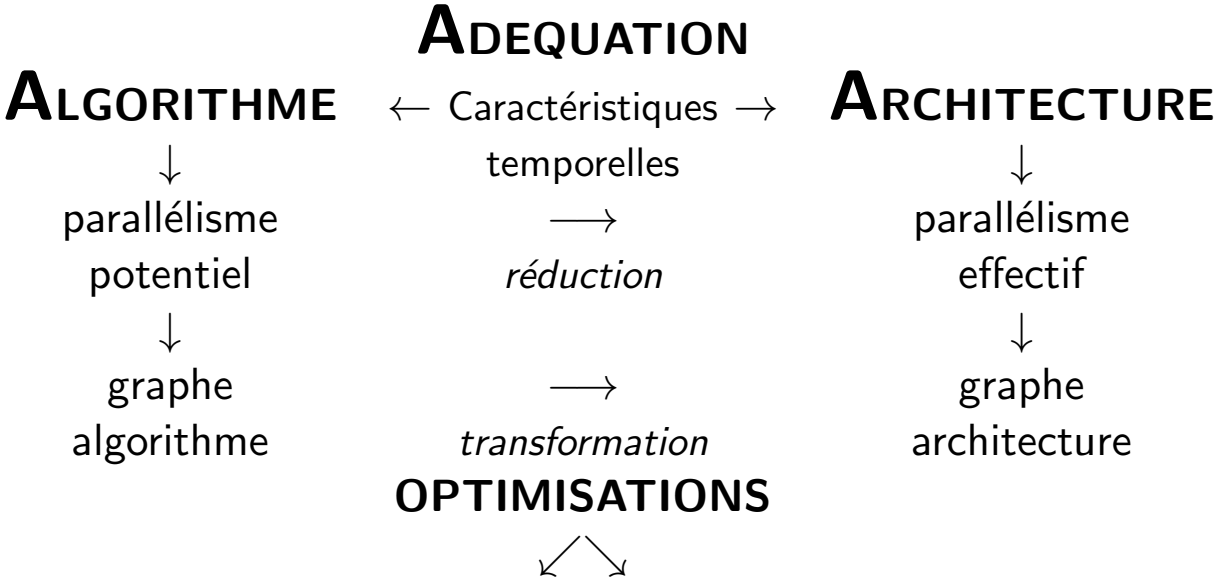
Pour un **graphe orienté** la relation A est antisymétrique, c.a.d. que si $s_1 A s_2$ et $s_2 A s_1$ alors $s_2 = s_1$, elle est transitive et elle n'est pas reflexive. C'est donc une **relation d'ordre strict** (notée $>$ différent de $\geq$ non strict).

Si l'ensemble des arcs est tel que $A \subset S \times S$ alors A est une **relation d'ordre strict partiel**, $\bar{A} = S \times S$ est une **relation d'ordre strict total**.

Un **stable** du graphe est un sous-ensemble de sommets qui ne sont pas deux à deux en relation A, en considérant que le graphe n'est pas orienté, c.a.d. un sous-ensemble de sommets qui ne sont pas deux à deux reliés par une arête, on ne considère pas l'orientation de l'arc. Le **plus grand stable** du graphe est noté A^*, on a $A \cup A^* = \bar{A}$.

Implantation optimisée : méthodologie AAA

Principes



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

CI spécifiques
NETLIST

Spécification des algorithmes

Généralités

Algorithme

La **spécification fonctionnelle** est réalisée en décrivant le système de commande sous la forme d'un **algorithme** qui devra être **implanté** sur des processeurs et/ou des circuits intégrés spécifiques interconnectés.

L'algorithme décrit, éventuellement de manière **hiérarchique**, les fonctions nécessaires à la réalisation de la spécification fonctionnelle ainsi que l'ordre partiel, dû aux dépendances de données, dans lequels ces fonctions vont s'exécuter. Il décrit aussi comment certaines fonctions seront **exécutées conditionnellement** ou bien seront **répétées un certain nombre de fois**.

Il y a **plusieurs approches** pour décrire un algorithme à l'aide de modèles formels fondés sur des graphes.

Modèle d'algorithme

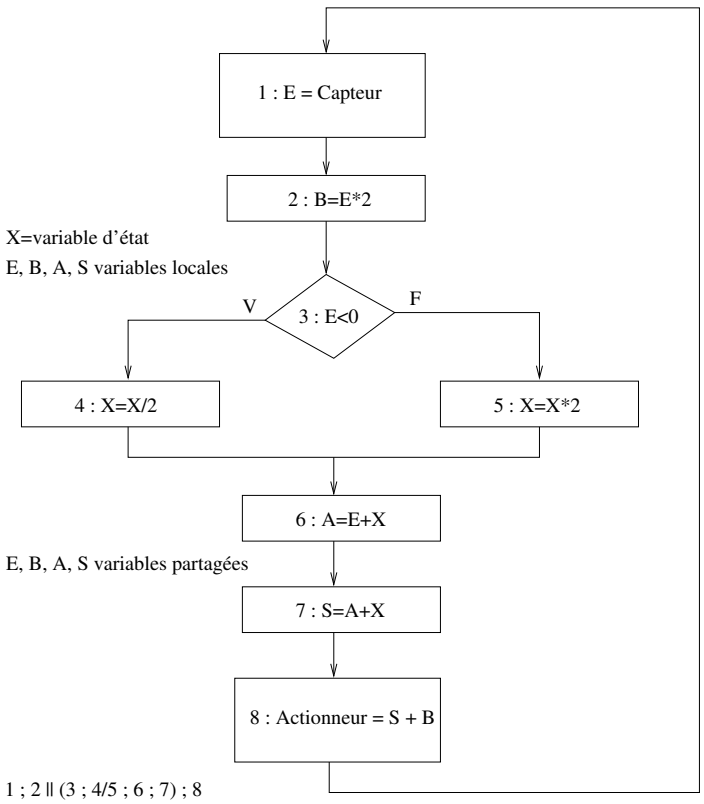
Graphes flot de contrôle 1/3

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

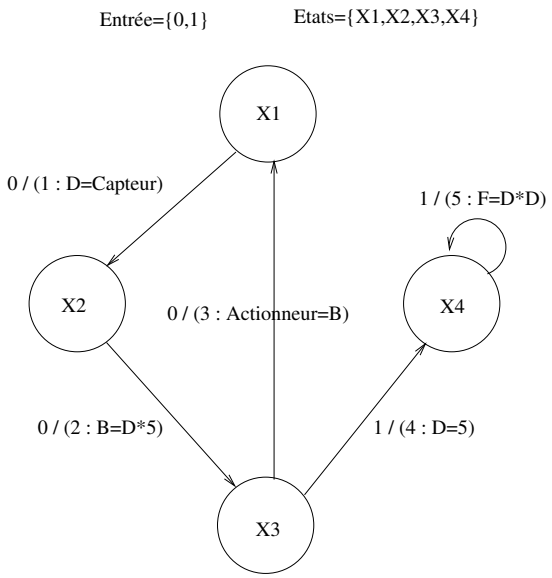
- un **organigramme** dont les **sommets** sont des opérations (fonctions) et les **arcs orientés** des **dépendances de contrôle** (branchement avant inconditionnel - exécution en séquence d'opérations - et branchement avant conditionnel sur test pour exécution alternative d'opérations appelé divergence en `OU`, branchement arrière pour boucle) qui induisent des précédences entre opérations. Une opération 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 (système réactif) est modélisée par une boucle (organigramme) et par l'arrivée des événements (automate).

Modèle d'algorithme

Graphes flot de contrôle 2/3



ORGANIGRAMME



D, B variables partagées

D variables réutilisées

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

AUTOMATE

Modèle d'algorithme

Graphes flot de contrôle 3/3

Dans ce modèle :

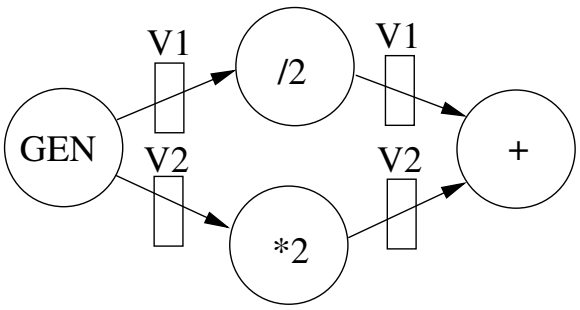
- les opérations accèdent aux données contenues dans des variables qui peuvent être **réutilisées et/ou partagées** par plusieurs opérations, ce qui est une source importante d'**erreurs**,
- l'ordre dans lequel les données sont accédées par les opérations et l'ordre dans lequel sont exécutées les opérations, 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.

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**. Une dépendance de contrôle induit une précédence entre opérations. Une opération est exécutée dès que toutes ses données d'entrées - produites par des opérations qui la précèdent - sont présentes, elle produit alors toutes ses données de sortie - consommées par des opérations qui la succèdent (règle d'activation de Milner). Les sommets sans prédécesseurs s'exécutent en premiers.



Modèle d'algorithme

Graphe flot de données 2/3

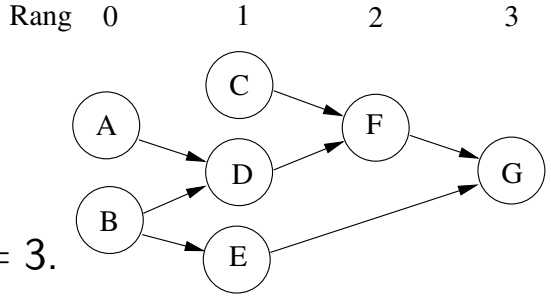
Dans ce modèle :

- à chaque arc correspond un transfert de donnée sans notion de variable (assignation unique) et donc de possibilités d'erreurs en les partageant, cependant les arcs pourront être implantés par des variables dont le partage et la réutilisation seront gérés **automatiquement** 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 précédence, le flot de contrôle et de données induit un **ordre partiel** sur l'exécution des opérations : divergence en `ET`,

Modèle d'algorithme

Graphe flot de données 3/3

- cet ordre partiel représente le **parallélisme potentiel** inhérent à la spécification de l'algorithme. Plus formellement c'est le **plus grand stable du graphe** dont on supprime les sommets qui sont en relation d'ordre par transitivité. On détermine dans cet ensemble 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 changer le rang d'un sommet pour augmenter le nombre de sommets d'un autre rang.
  Le maximum des cardinaux de ces sous-ensembles donne le nombre de processeurs que l'on peut mettre en **parallélisme potentiel**, ici $Max(Card\{A, B\}, Card\{C, D, E\}) = 3$.



- 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 proposé est un **graphe flot de données répété conditionné factorisé**, c.a.d. un graphe flot de données étendu :

- **infiniment répété** dont chaque répétition correspond à une interaction avec le processus (système réactif), chaque répétition 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** c.a.d. qu'un sommet hiérarchique du graphe peut se décomposer en plusieurs sommets dont un seul (divergence en `OU`) sera exécuté lors d'une répétition infinie en fonction de la valeur de l'entrée de conditionnement, extension du graphe flot de données dynamique (Buck 1993). Les entrées de conditionnement reçoivent des **dépendances de conditionnement**. Les sommets conditionnés correspondent à la notion de branchement conditionnel dans le modèle flot de contrôle (équivalent à `If...Then...Else...`) ;

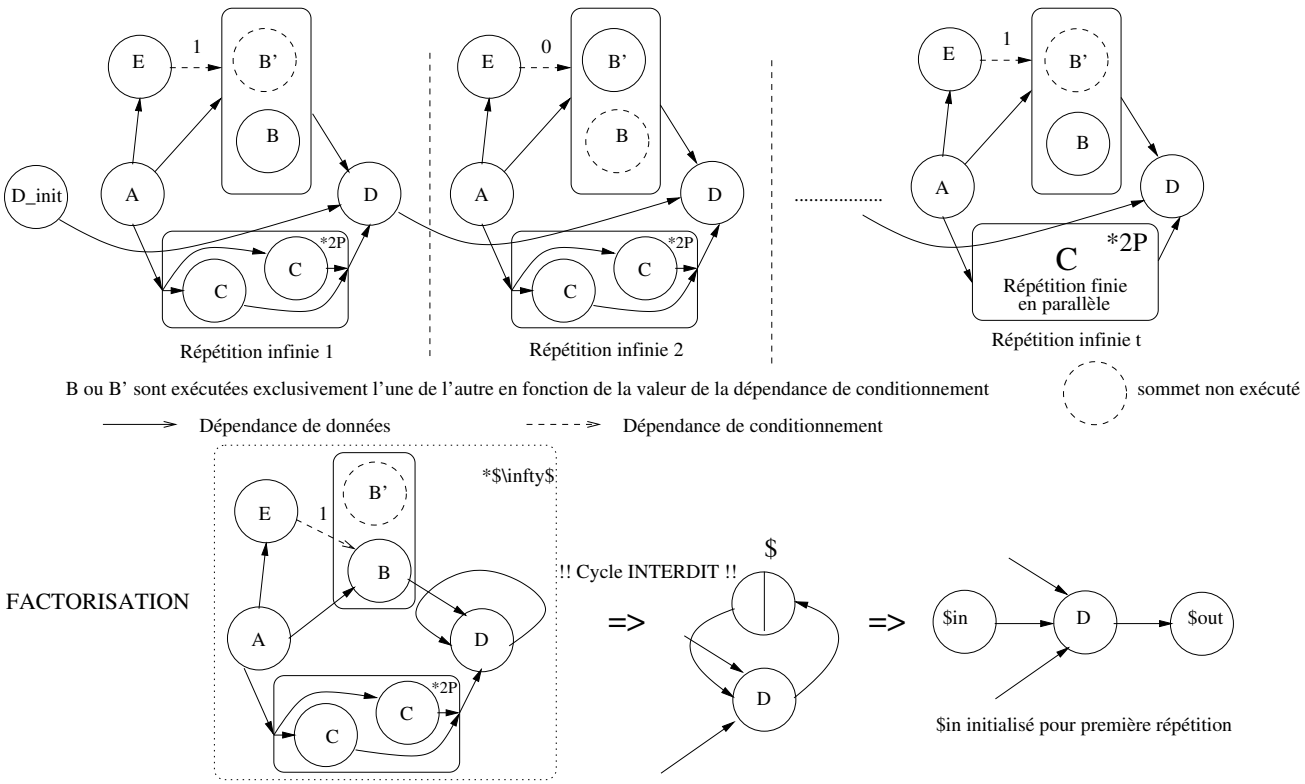
Modèle d'algorithme

Graphe flot de données répété conditionné factorisé 2/3

- **répété de manière finie** c.a.d. qu'un sommet hiérarchique du graphe peut se décomposer en plusieurs sommets identiques qui seront tous exécutés sur des données différentes, correspondant à du **parallélisme potentiel de données ou de flot**, par opposition au **parallélisme potentiel de contrôle** où les sommets sont différents, appelé par défaut **parallélisme potentiel**. Les sommets répétés de manière finie sont représentés par un seul sommet avec un indice de répétition en parallèle (P) (parallélisme de données) ou série (S) (parallélisme de flot). Ils correspondent à la notion de boucle ou d'itération dans le modèle flot de contrôle (équivalent à `For...To...Do...`) ;
- **factorisé** afin de simplifier le modèle, mais cela peut faire apparaître des cycles lorsque une opération de la répétition infinie t consomme des données produites à la répétition infinie $t - n$. Les cycles sont **interdits** pour garantir un comportement déterministe, sans interblocages (deadlocks). Chaque cycle **doit** contenir un **sommet $ (retard)** définissant **explicitement un des états de l'algorithme**.

Modèle d'algorithme

Graphe flot de données répété conditionné factorisé 3/3

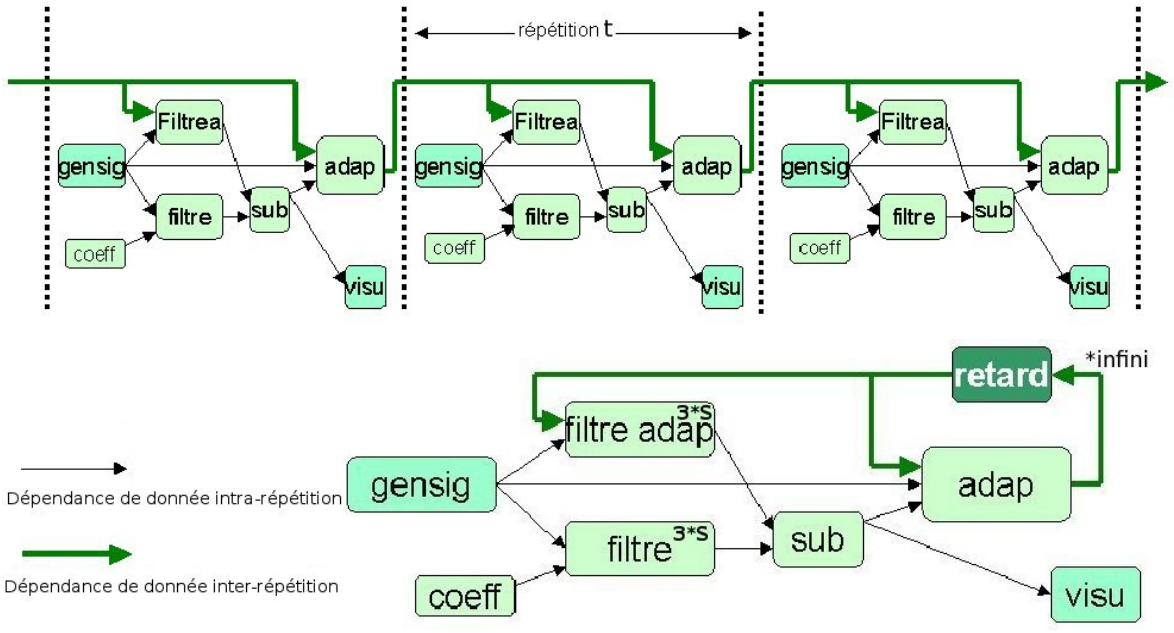


Chaque répétition infinie correspond à un instant logique t.

Modèle d'algorithme

Exemple : égaliseur adaptatif

Sortie capteur *gensig* filtrée par un filtre transversal à coefficients fixes et par un FIR dont les coefficients sont calculés par un algorithme adaptatif, la différence des sorties des filtres est visualisée par un actionneur *visu*.

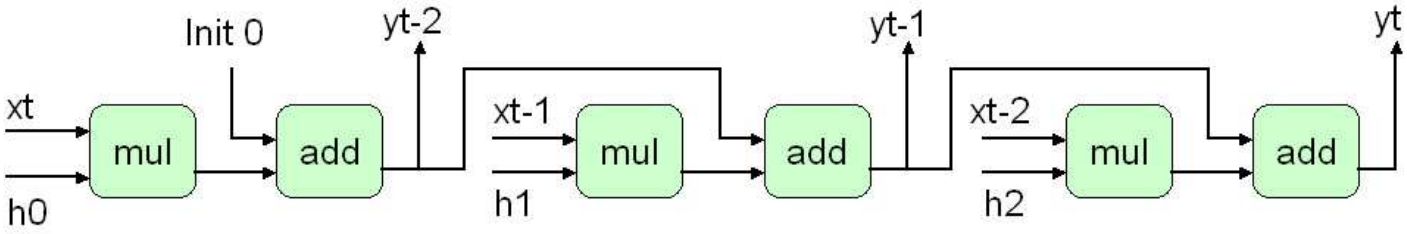


Modèle d'algorithme

Exemple : filtre transversal

Une entrée vecteur de 3 éléments contenant les coefficients (h_0, h_1, h_2), une entrée vecteur de 3 éléments contenant le passé de l'entrée (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.

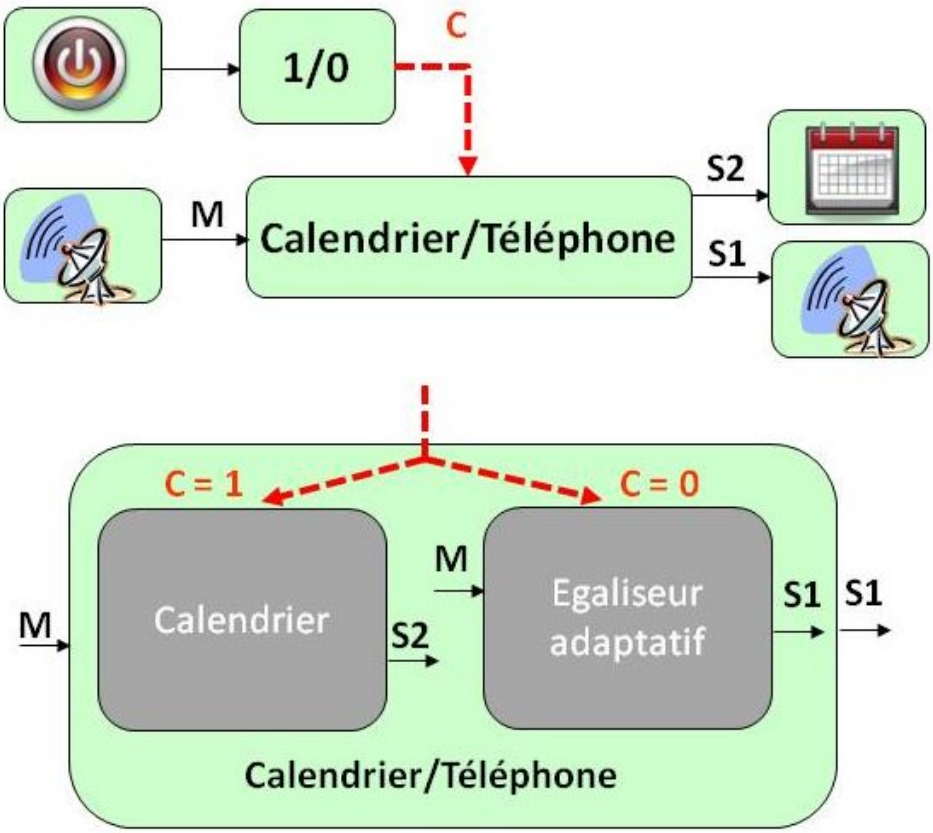


Fork : H(h0, h1, h2) et X(xt, xt-1, xt-2)

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

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 de manipuler le temps.

Langages de spécification fontionnelle

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

Issus des langages CSP, CCS, TLA, etc., les **langages synchrones** permettent d'exprimer du parallélisme potentiel (concurrence) et de manipuler le temps.

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 synthétisée.

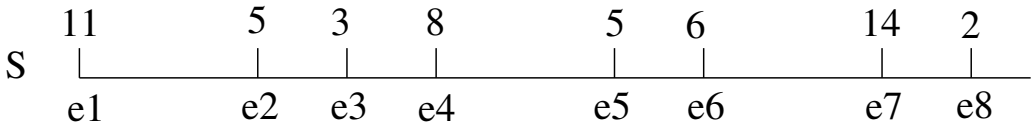
Langage synchrone SIGNAL

Relations entre signaux 1/4

SIGNAL est un langage **flot de données synchrone** permettant de spécifier des relations entre des **d'événements valués**, chacun d'eux appartenant à un ensemble infini d'événements appelé **signal** et prenant sa valeur dans un ensemble tel que les nombres réels, les nombres entiers, etc.

Un signal est associé à chaque entrée (resp. sortie) du système de commande correspondant a la sortie (resp. entrée) d'une opération capteur (resp. actionneur) et aussi à chaque entrée et à chaque sortie des autres opérations qui spécifent le système.
Il y a **quatre types** de relation :

1. une **relation de précédence** entre deux événements d'un même signal permettant d'associer une date à chaque valeur. C'est une relation d'ordre total strict représentée par un **diagramme temporel logique** :

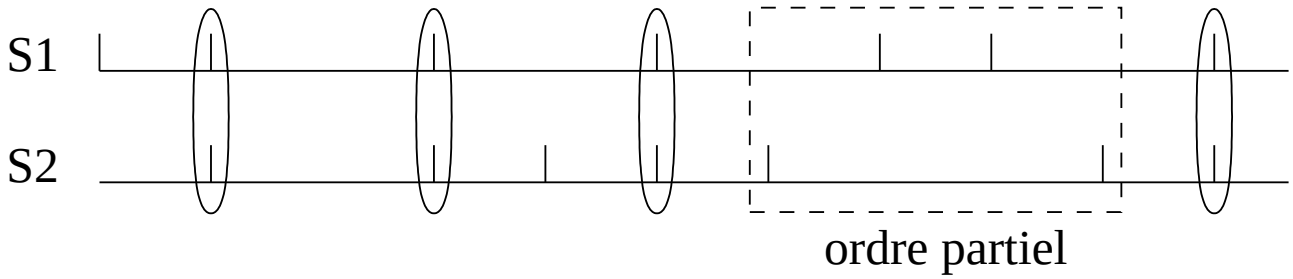


$$e1 < e2 < e3 < e4 < e5 < e6 < e7 < e8$$

Langage synchrone SIGNAL

Relations entre signaux 2/4

2. une **relation de synchronisme** entre deux événements de deux signaux différents. Lorsque deux événements sont **synchrones** on dit qu'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 (réflexive, symétrique, transitive).

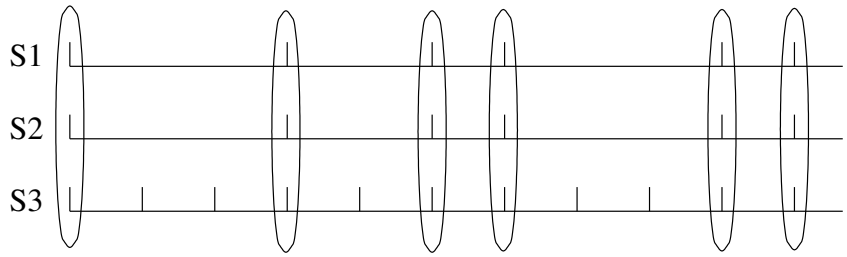


La relation d'équivalence de synchronisme entre événements de signaux différents combinée avec la relation d'ordre de précédence entre événements d'un même signal, ne permet pas de définir un ordre total sur les événements de signaux différents, mais seulement **un ordre partiel**.

Langage synchrone SIGNAL

Relations entre signaux 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** définissent une classe d'équivalence sur l'ensemble des signaux que l'on appelle **horloge**. On dit que ces signaux ont la même horloge. On définit une relation d'ordre totale sur les horloges. La plus grande des horloges définit le **temps logique** du système.

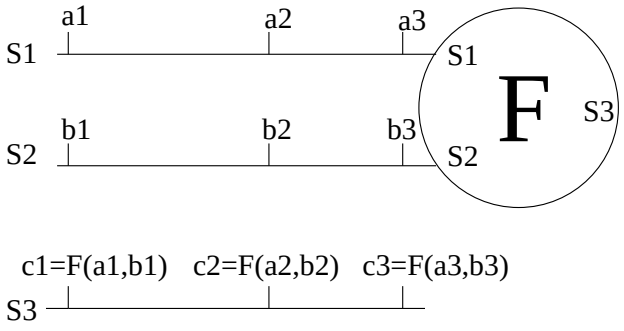


$S1$ et $S2$ ont la même horloge car ils sont synchrones. Certains événements de $S1$ et $S2$ sont absents relativement à $S3$ dont l'horloge est donc différente ce celle de $S3$ qui est ici la plus grande.

Langage synchrone SIGNAL

Relations entre signaux 4/4

4. des **relations d'entrée-sortie** définissant **4 types** d'instruction. Par exemple la **fonction immédiate** entre des signaux d'entrée et de sortie d'une opération, extension d'une fonction à des signaux.



Hypothèse des langages synchrones : les signaux de sortie d'une **fonction immédiate** sont synchrones avec les signaux d'entrée qui doivent aussi être synchrones. Cette fonction est **causale** dans l'instant logique. Une fonction n'est pas **causale** si une de ses sorties dépend d'elle même (cycle). **On ne prend pas en compte le matériel** au niveau de la spécification fonctionnelle. Le concept de durée n'existe qu'au travers du comptage des événements d'un signal.

Langage synchrone SIGNAL

Horloges des signaux

L'horloge d'un signal X est notée $P(X)$. C'est un ensemble 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 total** $\geq$.

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 $P(X) \geq T(X < 0)$.

Une horloges étant un ensemble de booléens, en notant $\cap$ par " " et $\cup$ par "+" on peut définir les relations de base sur les horloges :
$P(X) = P(Y)P(Z)$ $\quad$ $P(X) = P(Y) + P(Z)$ $\quad$ $P(X) \geq P(Y)$

Un programme ou **processus SIGNAL** correspond à la composition avec le caractère **"|"** 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 précédences qui définissent un ordre partiel d'exécution des instructions.

Langage synchrone SIGNAL

Quatre processus élémentaires

Syntaxe	Equation d'horloge	Relation E-S 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 si b présent vrai x type indifférent et b booléen
Mélange prioritaire y := x0 default x1	P(y) = P(x0) + P(x1)	y = x0 si x1 absent y = x1 si x0 absent y = x0 si 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       %erreur compilation car T(e<=0)#T(e>0)%
 |)
```

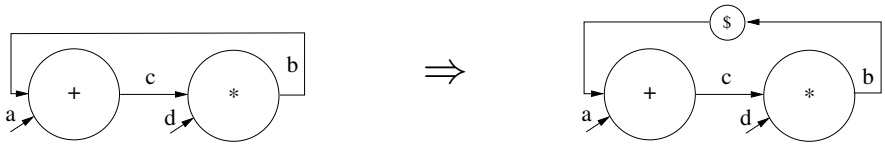
Langage synchrone SIGNAL

Graphe flot de données d'un processus

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

- chaque **sommet** comportant des **ports** d'entrée et de sortie représente un processus (élémentaire) SIGNAL auquel est associée une **équation d'horloge** définissant les horloges des sorties en fonction des horloges des entrées,
- chaque **arc** ou **dépendance de données**, entre un port d'entrée et un port de sortie, représente un signal auquel est associée une horloge,
- une suite d'arcs dont le premier sommet et le dernier sommet sont identiques conduit à un cycle dans le graphe et a un programme **non causal**. Les cycles sont **interdits**. Il faut y introduire un **retard** ;

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



- un processus élémentaire s'exécute en fonction de la présence des horloges de ses entrées définie par son équation d'horloge.

Langage synchrone SIGNAL

Fonctionnement du compilateur

Le compilateur effectue des **vérifications** :

- classiques sur valeurs (type, indices tableaux, division par zéro, etc.),
- formelles :
  - il vérifie qu'il n'y pas de cycles ne contenant pas un retard,
  - vérifie si le système d'équations d'horloges est correct, c.a.d. tente de déterminer les horloges des signaux de sortie en fonction des horloges des signaux d'entrée.
    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 aux signaux dont l'horloge est indéterminée en utilisant l'instruction de contrainte d'horloge : `^=`, à `X ^= Y` est associé `P(X) = P(Y)`.

Il génère un programme séquentiel, par exemple en C, qui permet de faire une **simulation fonctionnelle** et **temporelle logique** assurant un **ordre correct des événements** d'entrée et de sortie.

Langage synchrone SIGNAL

Diagrammes temporels logiques des signaux 1/3

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

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

Processus élémentaires ne modifiant pas les horloges

`X := A + B (X somme de A et B)`

A :	2	1	5	4	3
B :	2	1	5	4	3
X :	4	2	10	8	6

Langage synchrone SIGNAL

Diagrammes temporels logiques des signaux 2/3

`ZX := X $ 1 (Retard de 1 instant logique de ZX p.r. a 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 => P(X)=P(Y) (contrainte d'horloge) X, Y type event`
`L'horloge indeterminée de X prend l'horloge déterminée de Y`

Langage synchrone SIGNAL

Diagrammes temporels logiques des signaux 3/3

Processus élémentaires modifiant les horloges : $\perp$= absent

`X := A when B (sous-echantillonnage de A par B vrai)`

A :	1	2	3	4	$\perp$	5	6	$\perp$	7	8
B :	F	T	$\perp$	T	T	F	T	F	$\perp$	T
X :	$\perp$	2	$\perp$	4	$\perp$	$\perp$	6	$\perp$	$\perp$	8

`Y := X0 default X1 (melange de X0 et X1, X0 prioritaire)`

X0 :	1	3	$\perp$	5	6	$\perp$	8	9
X1 :	$\perp$	2	4	2	$\perp$	7	9	6
Y :	1	3	4	5	6	7	8	9

Langage synchrone SIGNAL

Signaux constants

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

Pas de problèmes avec la fonction immédiate et le `when` :

```
X := A + 1 => P(X)=P(A)=P(1)
X := 1 when B => P(X)=P(1)T(B) : X vaut 1 a l'horloge 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 a l'horloge de X0
Y := 1 default X1 => P(Y)=P(X1)=P(1) : Y=1 a l'horloge de X1
```

Langage 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**, notée additivement, permettant de définir la plus borne supérieure de deux horloges et d'**intersection**, notée multiplicativement, permettant de définir la borne inférieure de deux horloges, forment un treillis qui a les propriétés suivantes :

```
Commutativite : P(X)+P(Y) = P(Y)+P(X); P(X)P(Y) = P(Y)P(X)
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 :

```
P(X)+P(Y) = P(X) <=> P(X) >= P(Y)
P(X)P(Y)  = P(X) <=> P(Y) >= P(X)
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)
P(X)P(Y)=P(X) si P(Y) >= P(X);  P(X)+P(Y)=P(X) si P(X) >= P(Y)
```

Langage synchrone SIGNAL

Syntaxe des processus ou programmes 1/3

Les éléments terminaux du langage (mots-clés) sont soulignés, les éléments de syntaxe entre crochets "[. . .]" sont optionnels, "/" 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 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 synchrone SIGNAL

Syntaxe des processus ou programmes 3/3

% commentaires entre pourcent %

Dans une expression on peut combiner plusieurs processus élémentaires sachant que la fonction immédiate est plus prioritaire que le `when`, lui même plus prioritaire que le `default`.
Exemple :

```
y := a when b > 0 default z + 1 <=>
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 de données sont déduites des noms des signaux. Un signal de sortie est connecté au signal d'entrée de même nom, ou 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 synchrone SIGNAL

Exemple présentant la syntaxe

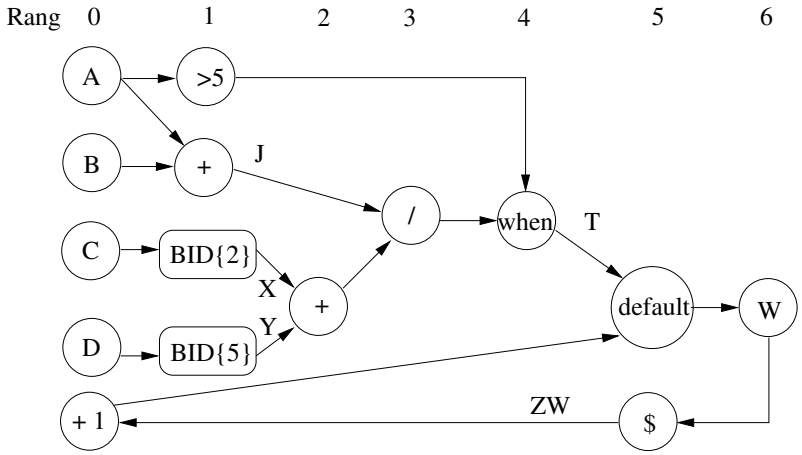
```
process EXEMPLE = {}
( ? integer A,B,C,D ! real W )
(| X := BID{2}(C)          %processus BID parametre 2 P(X)=P(C)%
 | J := A+B                %P(J)=P(A)=P(B)%
 | T := (X+Y)/J when A>5 %P(T)=P(A)T(A>5)=T(A>5)%
 | Y := BID{5}(D)          %processus BID parametre 5 P(Y)=P(D)%
                           %P(A)=P(B)=P(C)=P(D)%
 | ZW := W $ 1             %P(ZW)=P(W)%
 | W := T default ZW + 1 %P(W)=T(A>5)+P(W) => P(W)>=T(A>5) (1)%
 | W ^= A |)               %P(W)=P(A) verifie (1)%
where
 integer J, X, Y, ZW init 0 ; real T ;
 process BID = {integer PARAM}
  (? integer X ! integer Y )
  (| Y := X*PARAM |)
end
```

Un paramètre est un signal d'entrée **constant** interne/externe au processus

Langage 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 et actionneurs. **On ne considère pas les contraintes d'horloges**.

Parallélisme potentiel : $\{A, B, C, D, +1\}$ et $\{> 5, +, BID2, BID5\}$

Nombre de processeurs potentiellement en parallèle 5.

Langage synchrone SIGNAL

Exemples de processus 1/4

```
% Compteur sans fin sur le signal 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
```

Langage synchrone SIGNAL

Exemples de processus 2/4

```
% Memorisation d'un signal e a l'aide d'un autre signal m %
% la sortie s prend la valeur de l'entree e quand e %
% est present et prend la valeur precedente de s %
% quand e est absent %
% l'horloge de s doit etre plus grande que l'horloge %
% du signal de memorisation m %

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
```

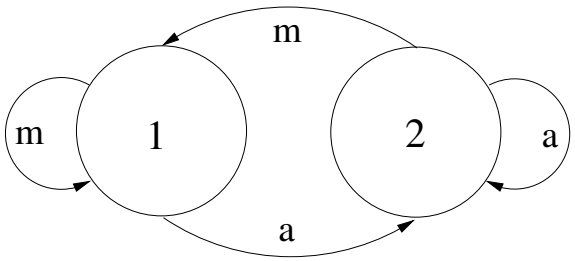
Langage synchrone SIGNAL

Exemples de processus 3/4

```
% Compteur sur le signal top remise a zero sur raz 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)=P(top)+T(raz)
 |)                                      verifie (1) %
where integer zn init 0
end
```

Langage synchrone SIGNAL

Exemples de processus 4/4




```
% Automate 2 etats marche (1) et arret (2), deux entrees %

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

Spécification des architectures multicomposant

Généralités

Architecture distribuée, parallèle, multiprocesseur, multicœur

Ces architectures sont formées de plusieurs processeurs, 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**. Quatre couples possibles :

- **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,
- **multicœur** : les processeurs sont sur un même circuit intégré (puce) et communiquent par mémoire partagée et/ou par un réseau sur puce.

Il existe différentes **topologies** : anneau, étoile, matrice, hypercube, complètement connecté, etc. Une **route** est une succession de couples (processeur, médium) connectés, se terminant par un processeur. Si un médium est un réseau de routeurs il faut donner la succession de routeurs.

Généralités

Parallélisme, architecture multicomposant

Types de parallélisme (classification de Flynn 1996) :

- **de contrôle** : (MIMD) les processeurs exécutent des traitements différents sur des données differentes, les dépendances le limitent,
- **de données** : (SIMD, SPMD) les processeurs exécutent le même traitement sur des données différentes,
- **de flot ou flux** : travail à la chaîne répétitif, pipe-line, (MISD) les processeurs exécutent des traitements différents sur la même donnée.

Architecture multicomposant : hétérogène, comprenant des processeurs de types différents, des circuits intégrés spécifiques de type différents, connectés par des média de communication de type différents et offrant des parallélismes de type différents.

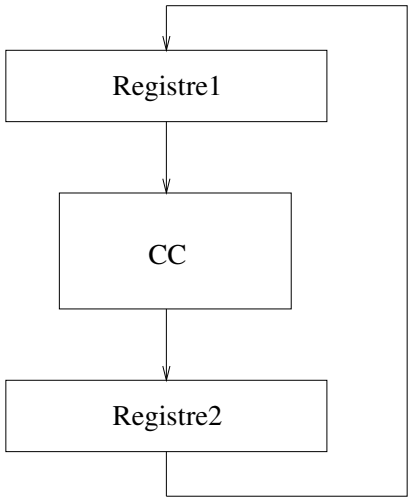
- **Processeur** : composant programmable qui exécute séquentiellement les instructions d'un programme.
- **Circuit intégré spécifique** (ASIC, FPGA) : composant non programmable qui exécute **une seule** opération.
- **Médium de communication** : point-à-point ou multi-point (bus).

Modèle d'architecture multicomposant

RTL

Modèle RTL : Register Transfert Level

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



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

Modèle d'architecture multicomposant

Machine séquentielle

Une architecture multicomposant est fondée sur la notion de machine séquentielle.

- **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 $t : E \times X \to X \qquad x_{t+1} = t(e, x_t)$

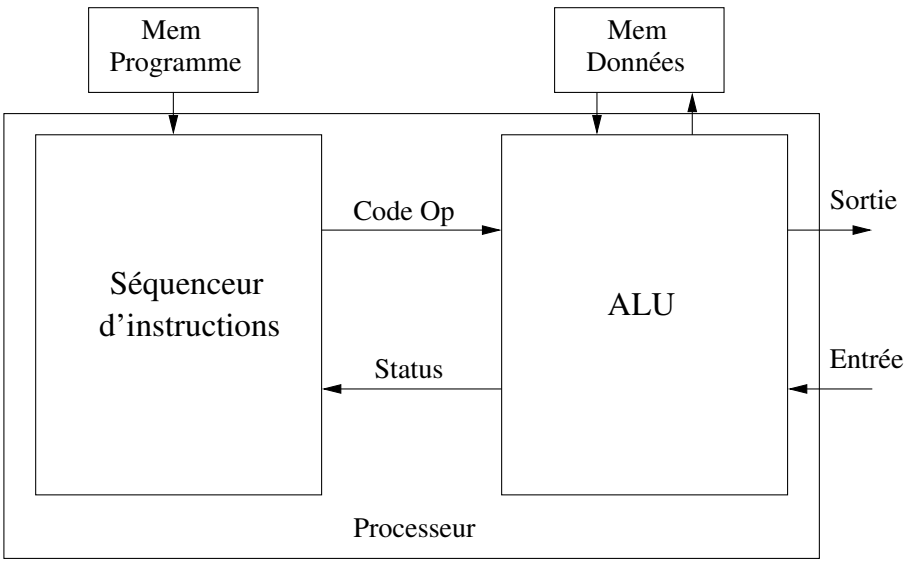
- **Automate fini avec sorties** (transducteur) : machine séquentielle dans le domaine de l'électronique numérique (E, X, i, f, t, S, s)

  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 multicomposant

Processeur

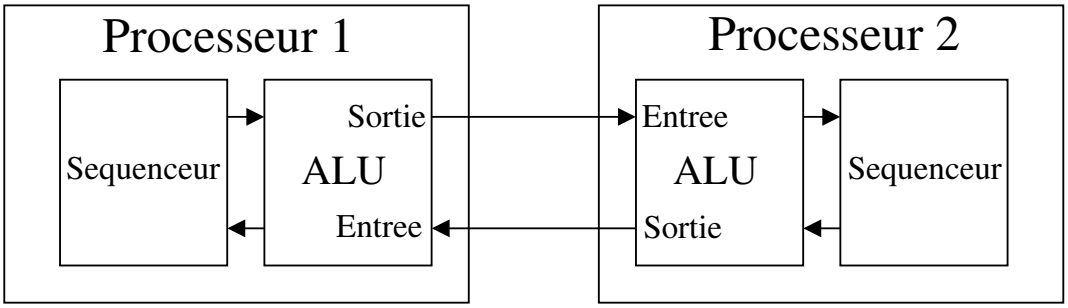
Processeur = 2 machines séquentielles connectées : séquenceur et ALU.
Le séquenceur lit l'état produit par l'ALU, lit une instruction dans la mémoire programme et écrit un code opératoire dans l'ALU.
L'ALU lit le code opératoire et exécute l'opération qui lit une donnée dans la mémoire de donnée et écrit un résultat dans la mémoire de donnée.



Modèle d'architecture multicomposant

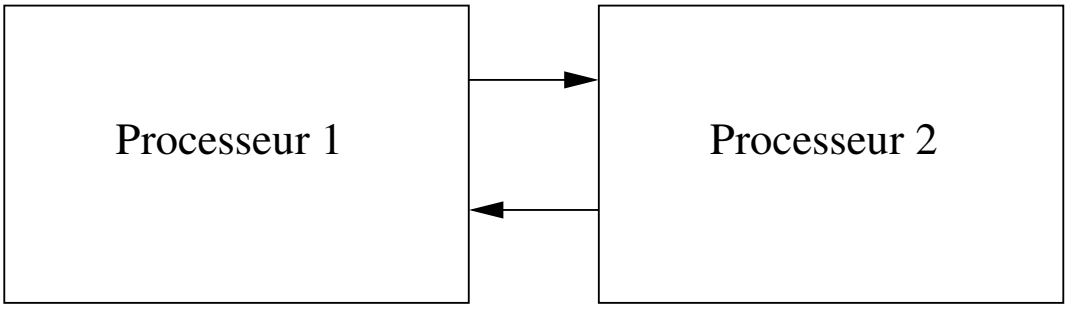
Modèle simple d'architecture distribuée ou parallèle

Deux processeurs, formés chacun de deux machines séquentielles, connectés par des liens point-à-point forment une architecture parallèle.



Abstraction

⇓



Modèle d'architecture multicomposant

Modèle multicomposant AAA 1/2

- **sommet** : machine séquentielle atomique de **quatre types** :
  1. **opérateur** séquence des opérations (ALU, FPU . . .) et des dépendances de données quand il n'y a pas de communicateurs,
  2. **communicateur** séquence des dépendances de données (DMA . . .),
  3. **mémoire** :
     - à accès aléatoire (RAM) :
       - de données (D) ou de programme (P),
       - de communications de données partagée par les communicateurs,
     - à accès séquentiel (SAM) : de communications de données, distribuées sur les communicateurs, par 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** : constituée de deux arcs de sens opposés, connexion bidirectionnelle entre sommets qui ne peut pas connecter des sommets de même type, sauf pour les sommets de type mux/demux.

Modèle d'architecture multicomposant

Modèle multicomposant AAA 2/2 et abstraction 1/2

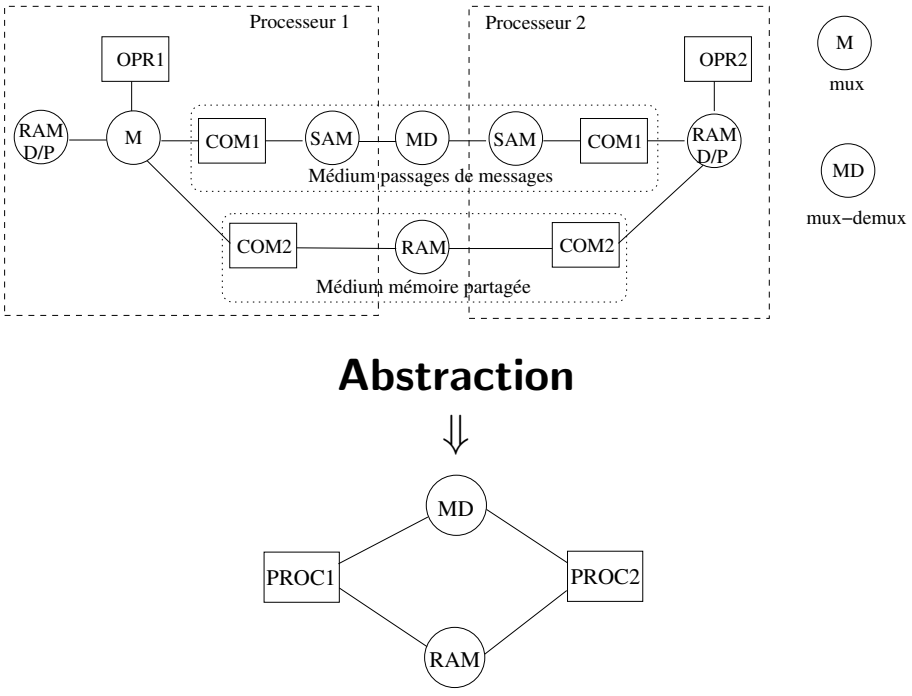
Processeur : graphe formé d'un seul opérateur et de plusieurs mux/demux, **RAM P**, RAM D, communicateurs, SAM.
Circuit intégré spécifique : graphe formé d'un seul opérateur et de plusieurs mux/demux, RAM D, communicateurs, SAM.
Médium passage messages : des communicateurs, SAM, mux, demux.
Médium mémoire partagée : des communicateurs, RAM.
Bus : élément passif constitué d'un mux et d'un demux permettant l'accès de plusieurs vers un (mux) ou de un vers plusieurs (demux).

Abstraction

Un sous-graphe processeur ou circuit intégré spécifique peut être **abstrait** en un sommet opérateur unique. Les mémoires programme et données, les mux et demux sont cachés. Un sous-graphe médium peut être **abstrait** en un sommet médium unique. Pour les communications par passage de messages chaque communicateur et sa SAM distribuée est représenté par un port de l'opérateur, le MD est conservé. Pour les communications par mémoires partagées seul chaque communicateur est représenté par un port.

Exemple de modèle d'architecture multicomposant

Abstraction 2/2

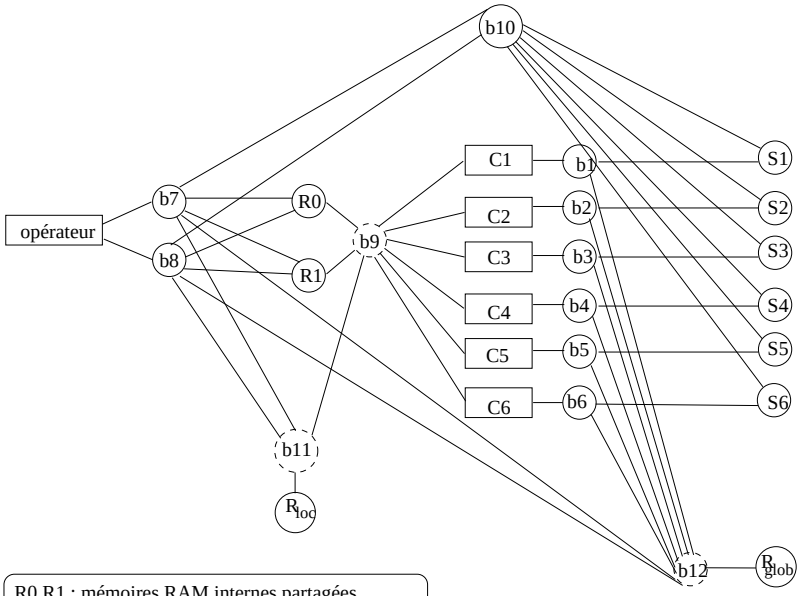


Abstraction

L'abstraction conduit à un graphe de taille plus faible donc à une optimisation plus rapide bien que moins précise.

Exemple de modèle d'architecture multicomposant

TMS320C40

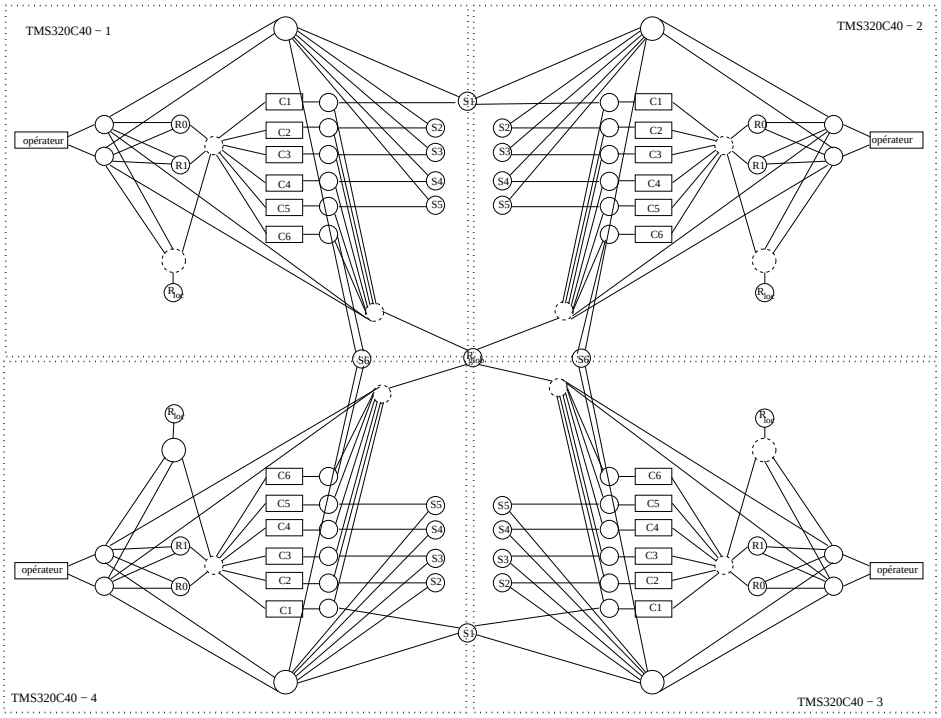


R0,R1 : mémoires RAM internes partagées
R_{loc},R_{glob} : mémoires RAM externes partagées
C0 à C6 : communicateurs
S1 à S6 : mémoires SAM
b1 à b8, b10 : demux
b9, b11, b12 : mux

29 sommets dans le graphe d'architecture

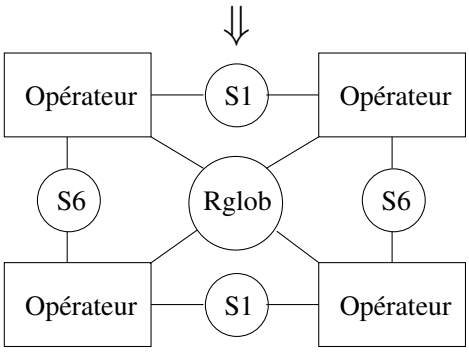
Exemple de modèle d'architecture multicomposant

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



109 sommets

Abstraction

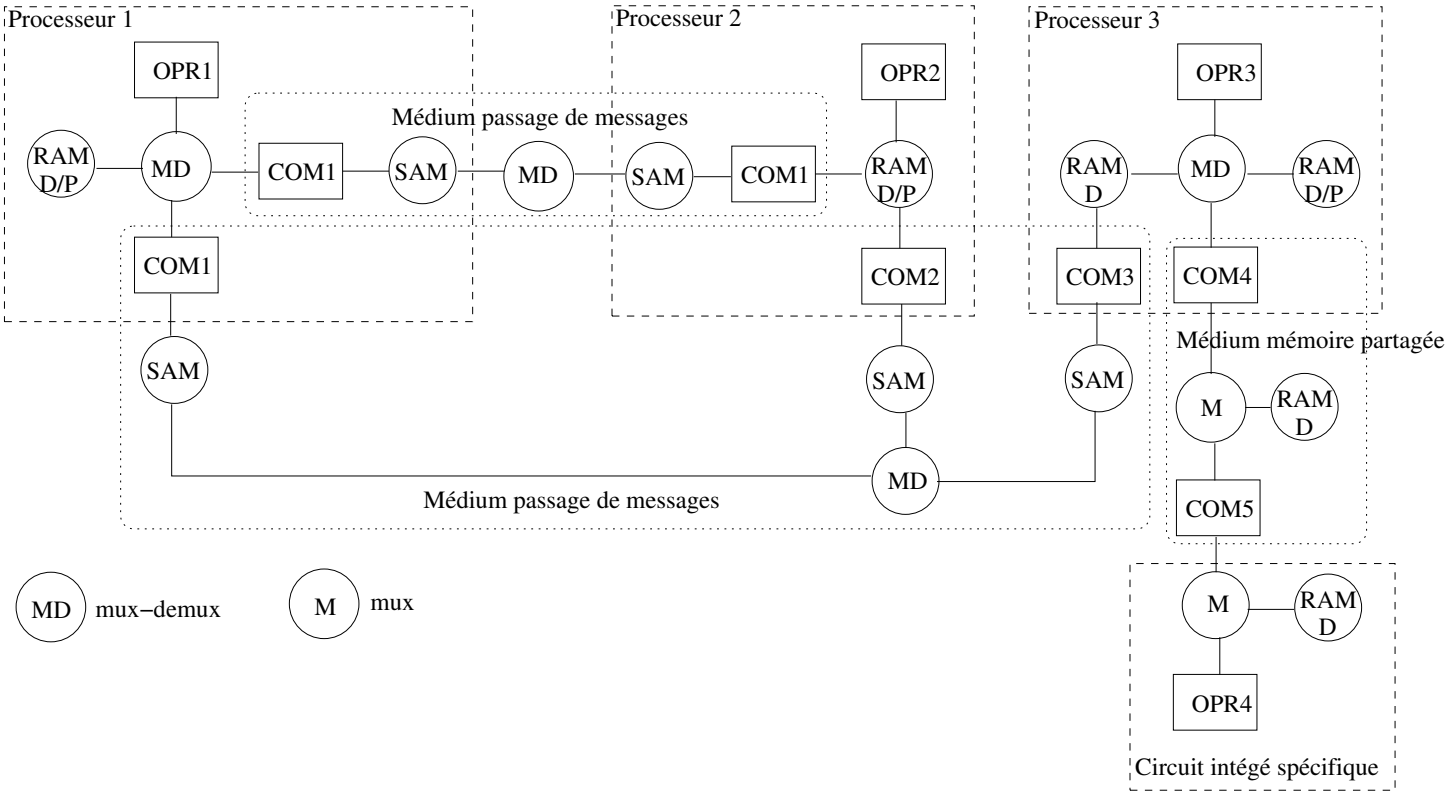


On cache mémoires (D/P) et mux/demux, les communicateurs sont les ports des opérateurs.

9 sommets

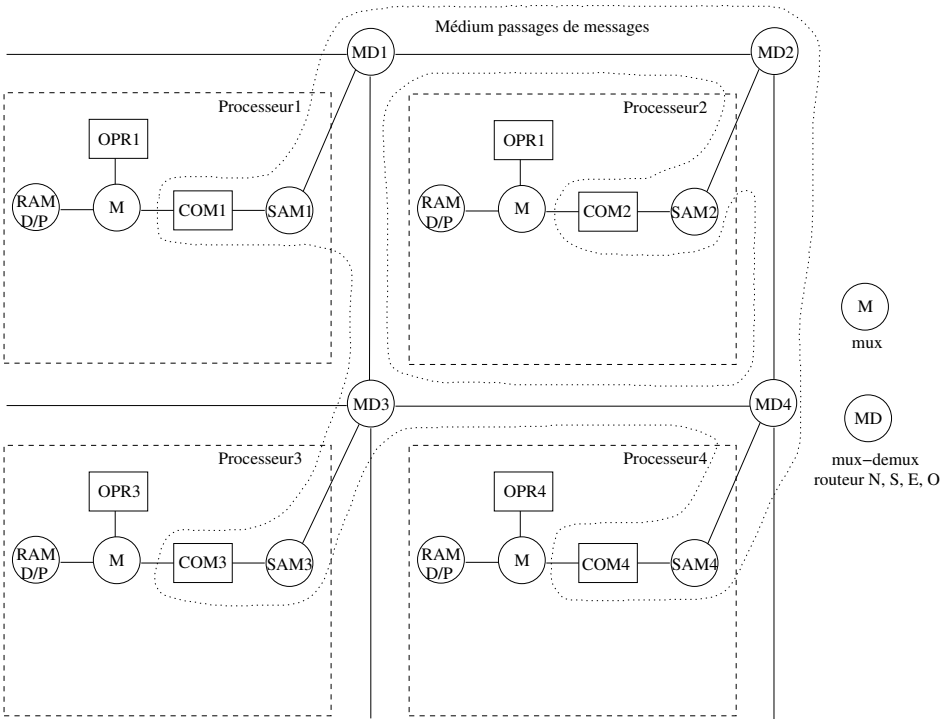
Exemple de modèle d'architecture multicomposant

Trois processeurs et un CI spécifique connectés en point-à-point et multi-point



Exemple de modèle d'architecture multicomposant

Quatre processeurs connectés par un réseau



Médium passage de messages $\{(com1, sam1), (sam1, md1), (com2, sam2), (sam2, md2), (com3, sam3), (sam3, md3), (com4, sam4), (sam4, md4), (md1, md2), (md1, md3), (md2, md4), (md3, md4)\}$

Implantation optimisée

Généralités

Distribution et ordonnancement 1/2

L'**implantation** s'effectue à partir des spécifications fonctionnelle et extra-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 aux opérateurs (ressources de calcul) et les dépendances de données aux média (ressources de communication). On appelle aussi la distribution : allocation, répartition, partitionnement, placement.

Pour chaque opérateur l'**ordonnancement** consiste à déterminer l'ordre dans lequel les opérations qui lui ont été allouées seront exécutées, et pour chaque médium l'ordre dans lequel les dépendances de données qui lui ont été allouées seront exécutées.

L'ordonnancement doit **conserver** l'**ordre partiel des dépendances** et assurer que les **contraintes temps réel** des opérations sont respectées.

Généralités

Distribution et ordonnancement 2/2

Distribution et ordonnancement peuvent s'effectuer **en-ligne** pendant l'exécution de l'application ou **hors-ligne** avant l'exécution de l'application.

Les approches en-ligne permettent de prendre en compte des opérations non prévues lors de la spécification, mais elles ont surcoût important et ne sont pas déterministes.

Les approches hors-ligne demandent une connaissance détaillée de l'algorithme, de l'architecture et des caractéristiques temporelles des opérations. Elles ne peuvent pas prendre en compte des opérations non prévues lors de la spécification, mais elles ont un surcoût peu important. Elles sont déterministes donc bien adaptées au temps réel critique.

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

Ordonnancement temps réel monoprocesseur

Classique 1/9

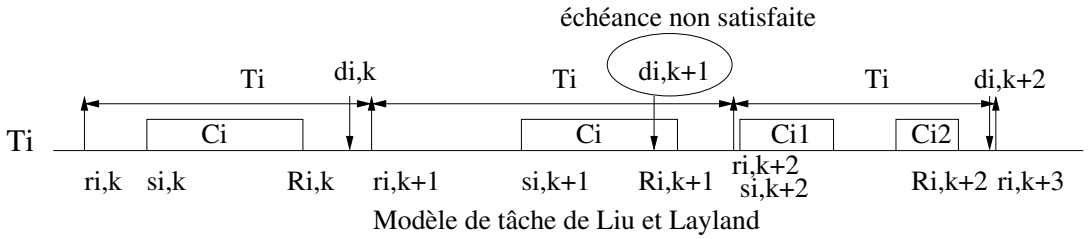
Le modèle de tâches **préemptives** classique (Liu et Layland 1973) est fondé sur l'utilisation d'un exécutif en-ligne pour lequel chaque tâche i est la répétition d'une "instance" ou "job" d'indice $k = 1..\infty$.

Une **tâche** est une **opération caractérisée temporellement** avec :

- **date d'activation** ou de réveil r_i^k (release time), éventuellement périodique (répétition infinie) de période T_i alors $r_i^k = r_i^0 + kT_i$,
- **première date d'activation** r_i^0 ou déphasage (offset),
- **date de début d'exécution** (start time) $s_i^k \neq r_i^k$,
- **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**,
- **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 échéance absolue et durée déjà exécutée.

Ordonnancement temps réel monoprocesseur

Classique 2/9



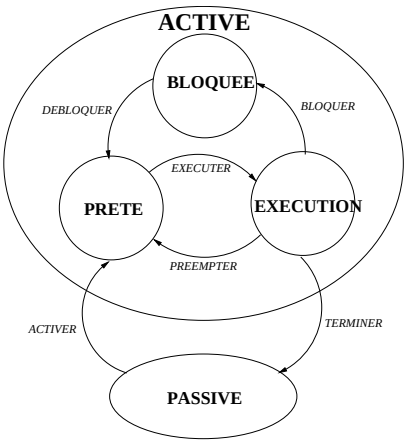
Modèle de tâche de Liu et Layland

Une tâche est **ordonnançable** si ses instances respectent leur échéance.
Un ensemble de tâches est **ordonnançable** si toutes ses tâches le sont.
L'**ordonnanceur** (scheduler) de l'exécutif suit un **algorithme d'ordonnancement temps réel** préemptif fondé sur des **priorités**.
La **préemption** autorise une tâche à être interrompue par une tâche plus prioritaire. Elle augmente le nombre d'ordonnancements possibles, mais il faut prendre en compte son **coût** dans celui de l'exécutif afin que les conditions d'ordonnançabilité soient **respectées**.
Les **priorités fixes**, utilisées par l'ordonnanceur pour faire le choix de la prochaine tâche à ordonnancer, ne changent pas au cours de l'exécution.
Les **priorités dynamiques** peuvent changer lorsqu'une tâche est activée ou se termine, conduisant à un coût de l'ordonnanceur plus élevé.

Ordonnancement temps réel monoprocesseur

Classique 3/9

L'ordonnanceur est constitué d'un automate par tâche et d'un gestionnaire des automates de tâches



L'automate d'une tâche a **quatre états** :

PRETE à être exécutée car elle vient d'être activée,
EXECUTION en cours d'exécution,
BLOQUEE 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,
EXECUTER, *PREEMPTER* : produits par le **gestionnaire**,
BLOQUER, *DEBLOQUER* : produits par le **gestionnaire**.

Ordonnancement temps réel monoprocesseur

Classique 4/9

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

Fonctionnement d'un ordonnanceur à priorités fixes

Sur l'événement *ACTIVER* d'une tâche, cette tâche passe de l'état **PASSIVE** à **PRETE**, 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 **EXECUTION**.

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 *PREEMPTER* pour son automate qui passe alors de l'état **EXECUTION** à **PRETE**, il produit un événement *EXECUTER* pour la tâche qui vient de s'activer qui passe de l'état **PRETE** à **EXECUTION** et enfin il lance son exécution.

Ordonnancement temps réel monoprocesseur

Classique 5/9

Sur l'événement *TERMINER*, produit par la tâche en cours, celle-ci passe de l'état **EXECUTION** à **PASSIVE**, le **gestionnaire compare** les priorités des tâches qui sont dans l'état **PRETE** et produit l'événement *EXECUTER* pour l'automate de la tâche la plus prioritaire qui passe de l'état **PRETE** à **EXECUTION**. 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 **EXECUTION** à l'état **BLOQUEE**. Il produit l'événement *DEBLOQUER* lorsqu'elle peut de nouveau y accéder. Elle passe alors de l'état **BLOQUEE** à l'état **PRETE**.

Ordonnancement temps réel monoprocesseur

L'analyse de **faisabilité** d'un ensemble de tâches temps réel consiste à trouver des conditions dans lesquelles toutes les tâches **satisfont** leurs contraintes. L'analyse d'**ordonnançabilité** consiste à trouver ce même type de conditions mais avec un algorithme d'ordonnancement donné.
Pour un ensemble de n tâches **préemptives périodiques indépendantes** dont le coût temporel 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 les **conditions d'ordonnançabilité** avec les **algorithmes d'ordonnancements** les plus connus :

- priorités fixes (ne changent pas pendant l'exécution des tâches) :
  - algorithme **RM** (Rate Monotonic) : priorité inversement proportionnelle à la période,
    les tâches sont ordonnançables **si** $U \leq n(2^{1/n} - 1)$, $D_i = T_i$,
  - algorithme **DM** (Deadline Monotonic) : priorité inversement proportionnelle à l'échéance,
    les tâches sont ordonnançables **si** $\Delta \leq n(2^{1/n} - 1)$, $D_i \leq T_i$,

Ordonnancement temps réel monoprocesseur

- priorités dynamiques (recalculées aux activations et terminaisons) :
  - algorithme **EDF** (Earliest Deadline First) : priorité à la tâche de plus petite échéance absolue, *dynamique entre instances, fixe dans l'instance*,
    les tâches sont ordonnançables **si et seulement si** : $U \leq 1$, $D_i = T_i$,
    les tâches sont ordonnançables **si** : $\Delta \leq 1$, $D_i \leq T_i$,
  - algorithme **LLF** (Least Laxity First) : priorité à la tâche de plus petite laxité, *dynamique entre instances, dynamique dans l'instance*,
    les tâches sont ordonnançables **si** mêmes conditions que EDF.

Lorsque les **tâches sont dépendantes** les dépendances sont dûes soit :
- à des **précédences seulement**, on peut alors se ramener au cas sans dépendance en ajoutant des contraintes supplémentaires conduisant à modifier les dates d'activation et les échéances,
- à des **transferts 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 l'échange des données en fonction des valeurs (plus petite ou plus grande) des périodes respectives de la tâche productrice et de la tâche consommatrice.

Ordonnancement temps réel monoprocesseur

Classique 8/9

La préemption peut produire un problème d'**inversion de priorité** lorsque **plusieurs tâches dépendantes partagent une donnée** protégée par une section critique. L'automate d'une tâche qui veut accéder à une donnée passe dans l'état **BLOQUEE** tant que celle-ci n'est pas disponible. Si plusieurs tâches sont dans cet état cela peut produire un problème d'**interblocage**. Deux protocoles permettent de régler ce problème :

- l'**héritage de priorité** (priority inheritance protocol) : 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) : 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

Classique 9/9

Pour un ensemble de **tâches apériodiques** les algorithmes EDF et LLF peuvent encore être utilisés.

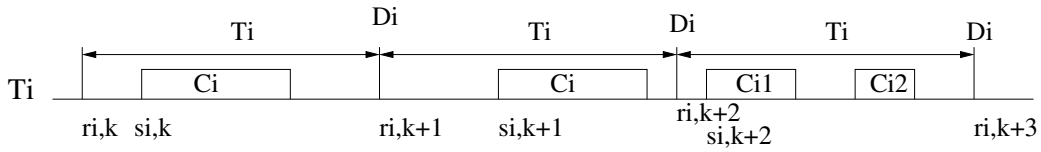
On peut aussi utiliser les algorithmes d'ordonnancement suivants :

- arrière plan : elles sont ordonnancées quand le processeur n'ordonnance pas d'autres tâches périodiques,
- serveur de tâches : elles sont ordonnancées par une tâche périodique supplémentaire de capacité égale à son WCET,
- vol de marge : elles sont ordonnancées pendant les laxités des autres tâches périodiques.

Ordonnancement temps réel monoprocesseur

AAA 1/3

Les applications temps réel **critiques** ont des tâches traitant les capteurs, les actionneurs et la commande qui doivent être sans gigue. C'est pourquoi elles doivent avoir une **période stricte** et être non préemptives afin que leur **temps de réponse** ne varie pas, étant alors égal à leur durée. Afin de simplifier le problème on considère que toutes les tâches sont de ce type et suivent le modèle **"AAA"** dans lequel une tâche o_i, appelée **opération**, se répète indéfiniment avec une **période stricte** T_i et une **durée d'exécution** C_i incluant le coût fixe de l'exécutif.



Modèle de tâche de Liu et Layland

Modèle d'opération AAA

Ordonnancement temps réel monoprocesseur

AAA 2/3

Dans ce modèle une opération o_i n'a **pas de date d'activation** mais seulement une **date de début d'exécution** $s_i^k = s_i^{k-1} + T_i = s_i^0 + k * T_i$ pour chaque instance k. Son **échéance relative est égale à la période** imposant que $C_i \leq T_i$.

Le coût de l'exécutif est constitué :

- du coût de l'ordonnanceur :
  - hors-ligne : lecture dans une table,
  - en-ligne : choix en comparant les priorités des tâches à chaque activation et terminaison de tâche,
- du coût des préemptions **pouvant entraîner d'autres préemptions**
  - hors-ligne : pas de préemptions,
  - en-ligne : sauvegarde et restitution de contexte.

On choisit donc d'être dans le cas **non préemptif**, même si cela complique les analyses d'ordonnançabilité et réduit le nombre d'ordonnancements possibles, car le coût des préemptions est nul. De plus on choisit d'utiliser un exécutif **hors-ligne** car le coût de l'ordonnanceur est plus faible et déterministe que pour un exécutif en-ligne.

Ordonnancement temps réel monoprocesseur

AAA 3/3

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 de données et **d'échéance égale à sa période stricte**.

Condition suffisante de faisabilité (Korst 1991 pour deux tâches, plus de deux tâches Kermia-Sorel 2009) : un graphe de n opérations o_i non préemptives dépendantes de durées d'exécution C_i et de périodes strictes T_i est ordonnançable **si** $\sum_{i=1}^{n} C_i \leq PGCD(T_i)$.

Ordonnancement temps réel multiprocesseur

Classique

Il y a 2 approches principales visant à minimiser le facteur d'utilisation U :

- **global :** un ordonnanceur unique pour tous les processeurs qui peut faire migrer les tâches d'un processeur à l'autre avec un coût très élevé pour les processeurs actuels. C'est donc un problème théorique, résolu avec l'algorithme optimal "Pfair",
- **partitionné :** un ordonnanceur par processeur sur chacun desquels on a distribué ou alloué (partitionné) des tâches qui doivent être ordonnançables. Minimiser le facteur d'utilisation est un problème **NP-difficile** équivalent à un problème de remplissage de boîtes à une dimension avec des objets de tailles différentes ("Bin Packing"). On ne peut le résoudre en un temps raisonnable que de manière approchée (non optimale) avec des **heuristiques** comme "First Fit", "Next Fit", "Best Fit", "Worst Fit", etc. Elles cherchent à distribuer les tâches sur les processeurs en vérifiant des conditions classiques d'ordonnançabilités (RM, EDF, etc). En général l'ordonnancement des communications est traité séparément sans considérer leur coût.

Ordonnancement temps réel multiprocesseur

AAA

A cause du coût prohibitif de la migration, on choisit l'**ordonnancement partitionné**. On cherche donc à résoudre un problème de distribution sur les différents processeurs et d'ordonnancement monoprocesseur non préemptif d'un ensemble d'opérations (tâches) devant, chacune, respecter ses **contraintes de dépendance de données** et **d'échéances égales à sa période stricte**. De plus on cherche à minimiser le temps d'exécution total en **considérant des temps de communication inter-processeur**.

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 cherchant parmi toutes les transformations, une transformation qui **minimise le temps d'exécution total** appelé **"latence de l'application"** = Max(latences entrée-sortie).

Formalisation de l'implantation AAA

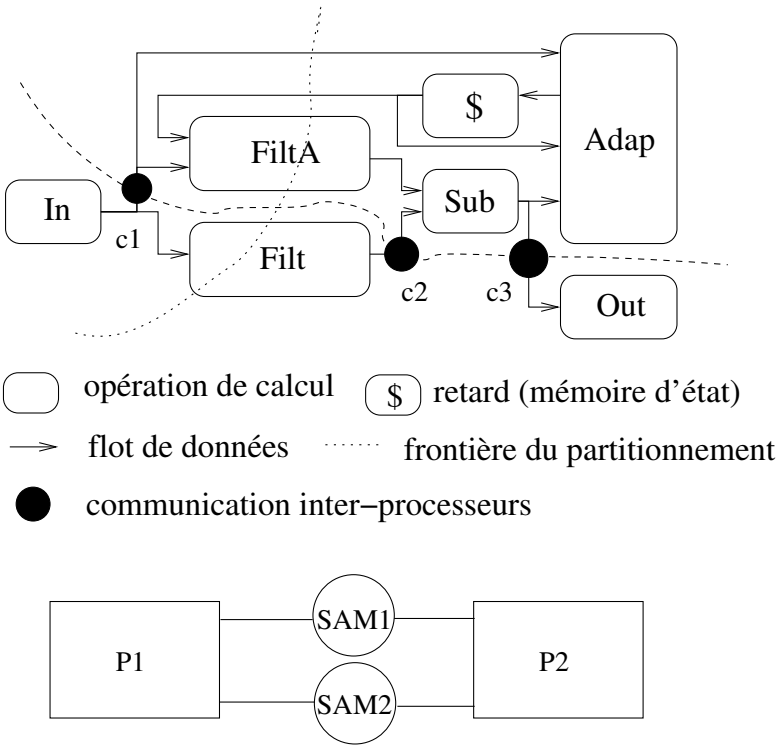
Transformations du graphe d'algorithme en fonction du graphe d'architecture

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 distribuées sur un même processeur mais qui ne sont pas reliées par des dépendances de données,
- **ajouter** des arcs de précédence entre opérations de communications distribuées sur un même médium mais qui ne sont pas déjà reliées par des précédences.

Formalisation de l'implantation AAA

Exemple de distribution et d'ordonnancement AAA



{In, Filt, Out} distribuées sur P1 et {FiltA, sub, Adap} distribuées sur P2.
{c1} distribuée sur SAM1 et {c2, c3} distribuées sur SAM2

Formalisation de l'implantation AAA

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 de données 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 grands mais il est fini**. Cette composition de relations **conserve les propriétés temporelles logiques** montrées lors des vérifications formelles.

Formalisation de l'implantation AAA

Relation de routage

Routage

Détermination de tous les chemins, suite de sommets reliés par des arcs, dans le graphe d'architecture (routes).

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

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

Formalisation de l'implantation AAA

Relation de distribution 1/2

Distribution

$\mathcal{O}$ = ensemble des opérations, $Card(\mathcal{O}) = n$
$\mathcal{D}$ = ensemble des dépendances de données, $d = (o, o'), o, o' \in \mathcal{O}$

Distribution des opérations sur les opérateurs = partition des n opérations de $\mathcal{O}$ 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.

$\mathcal{O} \supset \mathcal{O}_p$ = opérations exécutées par le processeur p
$\mathcal{D} \supset \mathcal{D}_p$ = dépendances entre opérations exécutées par p
$\mathcal{D} \supset \mathcal{D}_r$ = dépendances inter-partition

$$((\mathcal{O}, \mathcal{D}), (\mathcal{P} \cup \mathcal{L}, \mathcal{R})) \overset{distrib}{\longrightarrow} (\mathcal{G}_{d\mathcal{R}}, (\mathcal{P} \cup \mathcal{L}, \mathcal{R})) \quad \mathcal{G}_{d\mathcal{R}} = \left(\bigcup_{p \in \mathcal{P}} (\mathcal{O}_p, \mathcal{D}_p), \mathcal{D}_r \right)$$

Formalisation de l'implantation AAA

Relation de distribution 2/2

Distribution des opérations de communication sur les média = partition de $\mathcal{D}_r$ en Card($\mathcal{L}$) éléments dont le nombre est **calculable**.

Chaque dépendance inter-partition $(o_{i\,p_i}, o_{j\,p_j})$ 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 :

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

Un sommet o_l est une nouvelle **opération de communication** distribuée sur le médium l. Un arc $(o_{l_{k-1}}, o_{l_k}) = c_p$ est une dépendance de données distribuée sur le processeur p. On regroupe les o_l d'un même $l \in \mathcal{L}$ dans l'ensemble $\mathcal{O}_l$ et les c_p d'un même $p \in \mathcal{P}$ dans l'ensemble $\mathcal{C}_p$ avec $\mathcal{C}_p = \mathcal{C}_{p(calc,com)} \cup \mathcal{C}_{p(com,com)} \cup \mathcal{C}_{p(com,calc)}$

$$\mathcal{G}_{d\mathcal{R}} \xrightarrow{com} \mathcal{G}_{d\mathcal{L}} = \left(\bigcup_{p\in\mathcal{P}} (\mathcal{O}_p, \mathcal{D}_p \cup \mathcal{C}_p), \bigcup_{l\in\mathcal{L}} \mathcal{O}_l\right)$$

Formalisation de l'implantation AAA

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, un ordonnancement des opérations de calcul est un ordre total $\bar{\mathcal{D}}_p$ qui inclut l'ordre partiel $\mathcal{D}_p$, $\mathcal{D}_p \subseteq \bar{\mathcal{D}}_p$
De même sur chaque médium, un ordonnancement des opérations de communication est un ordre total $\bar{\mathcal{D}}_l$ qui inclut l'ordre partiel $\mathcal{D}_l, \mathcal{D}_l \subseteq \bar{\mathcal{D}}_l$

$$\mathcal{G}_{d\mathcal{L}} \xrightarrow{sched} \mathcal{G}_s = \left(\bigcup_{p\in\mathcal{P}} (\mathcal{O}_p, \bar{\mathcal{D}}_p \cup\ \mathcal{C}_p), \bigcup_{l\in\mathcal{L}} (\mathcal{O}_l, \bar{\mathcal{D}}_l)\right)$$

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 AAA

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 :

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

$$((\mathcal{O},\mathcal{D}),(\mathcal{P}\cup\mathcal{L},\mathcal{R})) \overset{distrib}{\longrightarrow} (\mathcal{G}_{d\mathcal{R}},(\mathcal{P}\cup\mathcal{L},\mathcal{R})) \overset{com}{\longrightarrow} (\mathcal{G}_{d\mathcal{L}},(\mathcal{P}\cup\mathcal{L},\mathcal{R})) \overset{sched}{\longrightarrow} (\mathcal{G}_s,(\mathcal{P}\cup\mathcal{L},\mathcal{R}))$$

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

Le nombre de partitions et le nombre d'ordre totaux tirés d'un ordre partiel étant calculables, **le nombre d'implantations possibles est calculable**.

Formalisation de l'implantation AAA

Loi de composition interne

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 $*$. Soit G_{al} l'ensemble des graphes d'algorithme et G_{ar} l'ensemble des graphes d'architecture, on a alors :

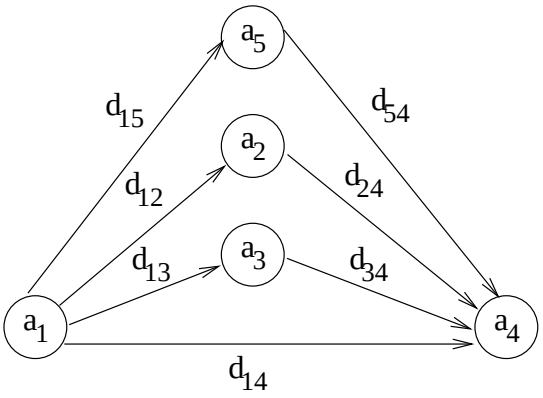
$$G_{al} \times G_{ar} \longrightarrow G_{al} \qquad g_{al} * g_{ar} = g'_{al}$$

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 et qui **n'introduisent pas de cycles** sur les sommets de calcul sans contenir un retard, ce qui créerait des interblocages. Les cycles sur les sommets de communication sont autorisés s'ils n'introduisent pas de cycles sur des sommets de calcul.

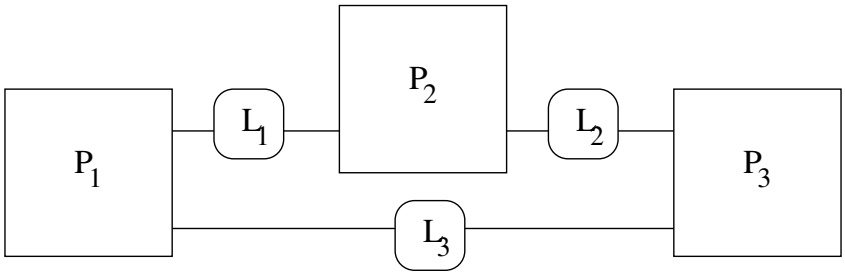
"**Compatible** avec l'ordre partiel du graphe d'algorithme" veut dire qu'aucun sommet ou arc n'a été supprimé, que seulement des arcs ont été ajoutés (ordre total) et que des sommets et des arcs ont été ajoutés sous la forme de chemins, uniquement, en remplacement des arcs inter-partition.

Formalisation de l'implantation AAA

Exemple de formalisation d'implantation AAA 1/3



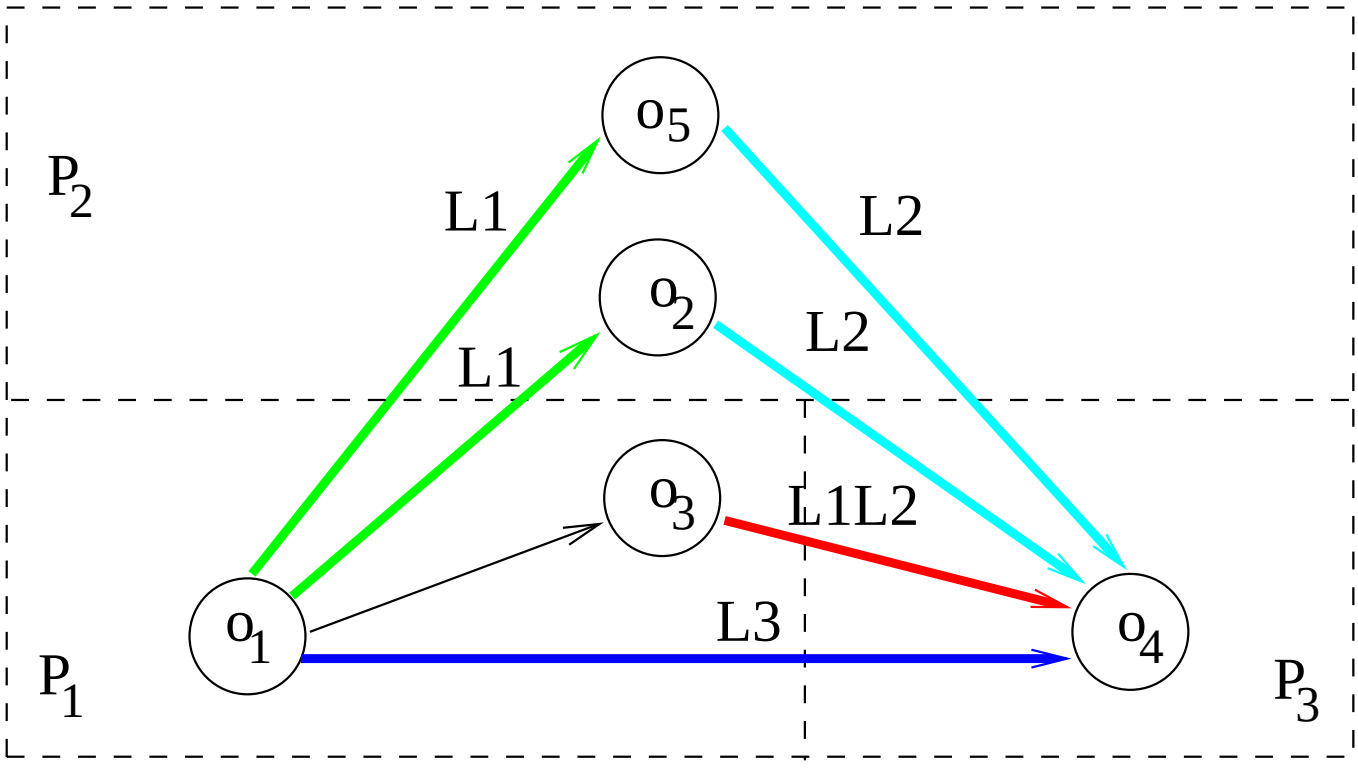
Graphe d'algorithme



Graphe d'architecture

Formalisation de l'implantation AAA

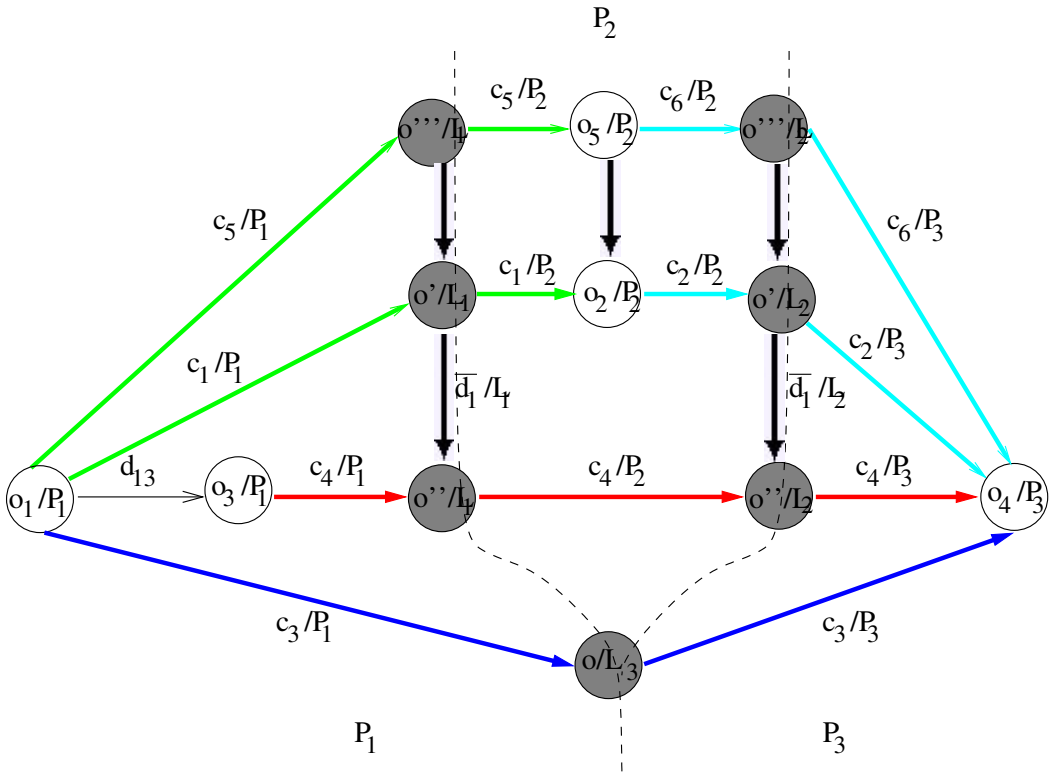
Exemple de formalisation d'implantation AAA 2/3



Distribution des opérations et des dépendances de données

Formalisation de l'implantation AAA

Exemple d'implantation AAA 3/3



Ordonnancement des opérations de calcul et de communication

Implantation optimisée : adéquation

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 de données** 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 données 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.

Implantation optimisée : adéquation

Principes 2/3

Le problème d'implantation **optimale** est aussi équivalent à un problème de "Bin Packing". Il 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 **N**on déterministe par un algorithme en temps **P**olynomial relativement à la taille du problème (donc NP ne vient pas de "Non Polynomial"). Ce sont des problèmes pouvant être résolus en énumérant toutes les solutions, chacune pouvant être testée par un algorithme en temps polynomial. La classe **P** contient les problèmes pouvant être résolus sur une machine de Turing déterministe par un algorithme en temps **P**olynomial. 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. Les problèmes NP-difficiles sont résolus optimalement en **temps exponentiel**.

Pour les problèmes de petites tailles, on peut utiliser des algorithmes exacts comme la programmation dynamique, la programmation linéaire en nombres entiers, les algorithmes d'évaluation séparation (B & B, B & C, etc.), la programmation par contraintes, etc.

Implantation optimisée : adéquation

Principes 3/3

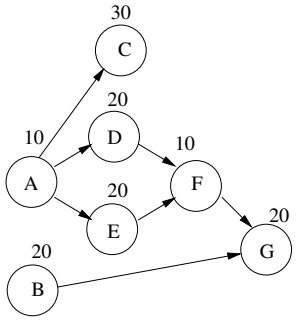
Pour les problèmes de tailles réalistes, on utilise des **heuristiques** qui donnent des solutions, empiriquement proches de l'optimal. En revanche les **algorithmes d'approximation** trouvent des solutions proches de l'optimal d'un facteur ϵ.
Il y a deux types d'heuristiques :

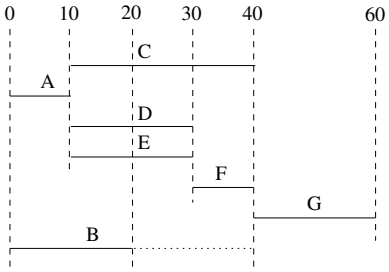
- sans retour arrière : très rapides, dites "**gloutonnes**". Elles ne partent pas d'une solution initiale et cherchent à chaque étape une solution localement optimale pour construire une solution finale qui n'est pas nécessairement globalement optimale. Elles font des choix déterministes généralement 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 un "**voisinage de solutions**". Elles peuvent faire des choix non déterministe pour sortir de minima locaux. Elles donnent des résultats empiriquement plus proches de la solution optimale que les heuristiques gloutonnes. Comme elles résolvent des problèmes génériques on les appelle "**métaheuristiques**" : "recuit simulé", "recherche tabou", "génétique" ou "colonie de fourmis", etc.

Implantation optimisée : adéquation

Chemin critique

La résolution du problème d'implantation avec minimisation de la latence est fondée sur le calcul du **chemin critique** (CC) du graphe d'algorithme étiqueté uniquement avec les durées des opérations de calcul. On ne tient pas compte du coût des communications. A chaque opération est associée un segment de longueur égale à sa durée qui est calé selon sa **date de début au plus tôt** déterminée par ses prédécesseurs et on détermine sa **date de début au plus tard** en fonction de ses successeurs. Un CC correspond à un chemin d'opérations sans **marge d'ordonnancement**, c.a.d. telle que chaque opération a sa date de début au plus tôt égale à sa date de début au plus tard. S'il y a plusieurs CC on prend le Max.





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

Implantation optimisée : adéquation

Caractérisation des opérations et des dépendances de données

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

Implantation optimisée : adéquation

Heuristique pour minimiser la latence = la cadence = la période stricte 1/4

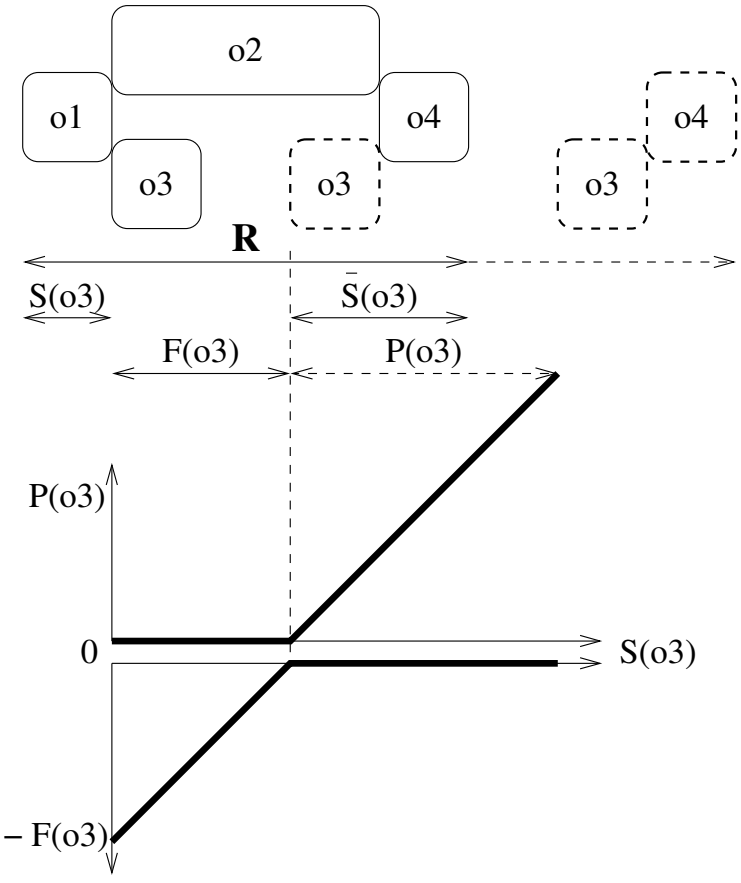
On utilise une **heuristique "gloutonne"** de distribution et d'ordonnancement rapide car de faible complexité $Card(\mathcal{O})Card(\mathcal{P})$.

Afin de respecter l'ordre partiel défini par les dépendances de données entre opérations du graphe d'algorithme, elle choisit à l'étape i une opération dans le sous-ensemble des opérations dont les prédécesseurs ont été distribués et ordonnancés, appelé **"candidats"**, qui optimise une fonction de coût locale. L'opération ainsi choisie est enlevée de l'ensemble des opérations à étudier. La fonction de coût appelée **pression d'ordonnancement** (schedule pressure) est telle que $\sigma(o, p) = P - F$:

- F différence entre date d'exécution au plus tôt et date d'exécution au plus tard : **marge d'ordonnancement** (schedule flexibility),
- P allongement du **chemin critique** causé par la prise en compte des coûts de communication : **pénalité d'ordonnancement** (schedule penalty) correspondant à un temps d'exécution partiel ou une latence partielle.

Implantation optimisée : adéquation

Heuristique pour minimiser la latence = la cadence = la période stricte 2/4



o1 → (o2, o3) → o4

$S(o3)$ date début plus tôt o3
$\overline{S}(o3)$ date début plus tard o3
$E(o3) = S(o3) + C$ date fin plus tôt
$F(o3)$ marge de o3
R latence partielle

La date de début au plut tôt de o4 est repoussée à cause de la durée de la communication due à o1 → o3 qui est supérieure à la marge d'ordonnancement $F(o3)$ que possède o3. Ceci rallonge la latence partielle R.

Implantation optimisée : adéquation

Heuristique pour minimiser la latence = la cadence = la période stricte 3/4

Les **candidats** $\mathcal{C}_i \subseteq \mathcal{O}$ à l'étape i sont le sous-ensemble d'opérations de $\mathcal{O}$ dont les prédécesseurs ont tous été distribués et ordonnancés.

HEURISTIQUE AAA MONO-PERIODE

1: $i = 0$, $V_0 = \mathcal{O}$ opérations du graphe d'algorithme

Tant que $V_i \neq \emptyset$

$i = i + 1$

Pour tout $o_j \in \mathcal{C}_i \subset V_i$

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

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

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

calculer latence partielle, $V_i = V_{i-1} - \{o_j\}$

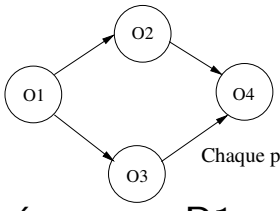
Fin tant que

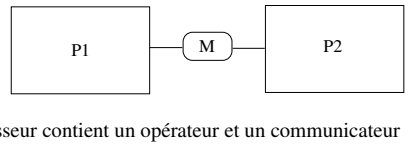
Si latence (dernière latence partielle) $\leq$ contrainte de latence **Fin**

Sinon modifier les contraintes de distribution/ordonnancement et/ou augmenter le parallélisme potentiel de l'algorithme $\mathcal{O} = \mathcal{O}'$ **Go to** 1

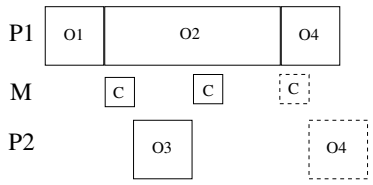
Implantation optimisée : adéquation

Heuristique pour minimiser la latence = la cadence = la période stricte 4/4





Chaque processeur contient un opérateur et un communicateur



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

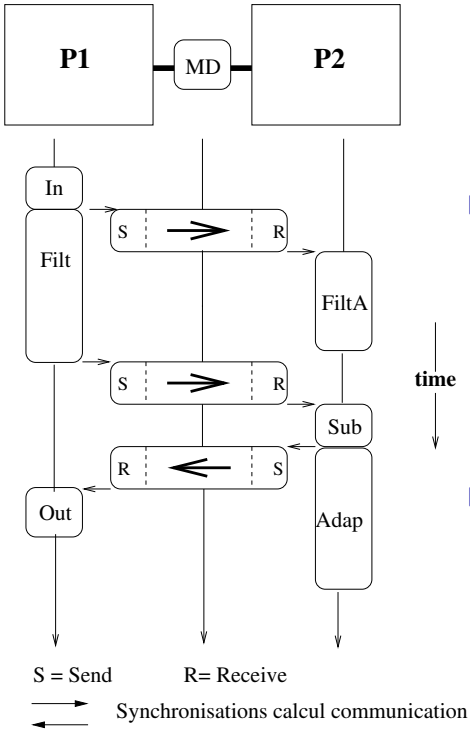
i	(o, p)	$\sigma(o, p)$	$\min \sigma$	$\max(\min \sigma)$	$V_0 = \{o1, o2, o3, o4\}$
1	(o1,P1)	10	(o1,P1)	(o1,P1)	$V_1 = \{o2, o3, o4\}$
2	(o2,P1)	10+30=40	(o2,P1)	(o2,P1)	$V_2 = \{o3, o4\}$
	(o2,P2)	(10+5)+30=45			
	(o3,P1)	10+10=20	(o3,P1)		
	(o3,P2)	(10+5)+10=25			
3	(o3,P1)	40+10=50			
	(o3,P2)	(10+5)+10=25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

Monoprocesseur 60, CC = 50, latence = 50, accélération maximale sans comm. = 60/50 = 1.2, accélération avec comm. = 60/50 = 1.2

Implantation optimisée : adéquation

Résultat de l'adéquation : table d'ordonnancement

Egaliseur implanté sur deux processeurs avec chacun un **opérateur** et un **communicateur** fonctionnant en parallèle et 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 initial de l'algorithme qui décrit une **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, le graphe d'implantation donne les **dates de début** et de **fin** des opérations de calcul et de communication. Leur longueur est proportionnelle à leur durée.

Implantation optimisée : adéquation

Heuristique multi-période stricte minimisant la latence

A chaque opération o_i de l'algorithme est associée une **période** stricte T_i, indépendante du processeur P_j, en plus d'une durée C_i dépendante du processeur P_j. On appelle **hyperpériode** le PPCM de ces périodes T_i.

`HEURISTIQUE AAA MULTI-PERIODE`

1 `Assignation` : **Pour toute** opération o_i
Pour tout processeur P_j
Si o_{iP_j} ordonnançable **Alors** $j+1$ **Sinon** `Fin`
(condition suffisante de faisabilité : $\sum_{i=1}^{n} C_{iP_j} \leq PGCD(T_i)$)

2 `Déroulement` : **Pour toute** opération, la dupliquer autant de fois que le rapport de l'hyperpériode sur la période de l'opération et ajouter les arcs supplémentaires nécessaires

3 `Ordonnancement` : **Pour toute** opération assignée et déroulée sur un processeur, calculer sa date d'exécution au plus tôt en tenant compte du coût des communications
Si une opération à été assignée à plusieurs processeurs
Alors choisir celui minimisant la latence

Implantation optimisée : adéquation

Heuristique pour minimiser 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

$$\textbf{Accélération maximale} = \frac{\text{somme des durées de toutes les opérations}}{\text{durée chemin critique sans communications}}$$

$\lceil \text{Accélération maximale} \rceil = [Card(\mathcal{P})] =$ Nombre maximal de processeurs identiques en **parallélisme effectif** nécessaires pour exploiter le **parallélisme potentiel** en tenant compte des durées.

processeurs en parallélisme effectif $\leq$ processeurs en parallélisme potentiel

Implantation optimisée : adéquation

Heuristique pour minimiser le nombre de processeurs

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

`META-HEURISTIQUE AAA`

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

Génération de code

Généralités 1/2

Les systèmes étant spécifiés fonctionnellement selon une approche LTT on **privilégie les implantations sur des architectures TT** (capteurs interrogés périodiquement) plutôt que des architectures ET (capteurs produisant des interruptions). La lecture des capteurs, l'exécution d'une répétition infinie de l'algorithme et l'écriture sur des actionneurs, peut être déclenchée :

- à l'aide d'une base de temps extérieure périodique,
- **auto-déclenchés** à l'aide d'une boucle de durée connue.

On se place maintenant dans ce dernier cas.

Sur chaque processeur le code de l'**exécutif** est **hors-ligne** si les choix d'optimisation et les décisions sont effectués avant l'exécution et **en-ligne** si les choix sont faits pendant. Pour les systèmes temps réel durs on **privilégie l'approche hors-ligne** qui est cohérente avec l'approche TT.

L'exécutif lit la **table d'ordonnancement** pour exécuter la boucle contenant une séquence d'opérations, d'attentes et de communications.

Génération de code

Généralités 2/2

- Le **code de l'exécutif** est constitué d'instructions système qui contrôlent (ordonnancement, conditionnement, répétition) le **code applicatif** associé aux opérations spécifiées dans l'algorithme.
- Le code de l'exécutif est automatiquement **synthétisé sur mesure**. Il tire parti au mieux des caractéristiques de l'application et induit ainsi un faible surcoût facile à déterminer ce qui le rend déterministe.
- 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 dynamique et a donc un surcoût plus important difficile à déterminer, mais a l'avantage d'être "standard".
- Le coût du code de l'exécutif doit être pris en compte le plus précisément possible pour que l'analyse d'ordonnançabilité soit fiable.
- On génère automatiquement pour chaque processeur un **pseudo code** afin qu'il soit **indépendant de l'architecture**, appelé **macro-code**, qui est formé d'un macro-exécutif et de macros applicatives.

Génération de code

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

Chaque **macro-code** est formé d'une **boucle infinie** ordonnançant des :

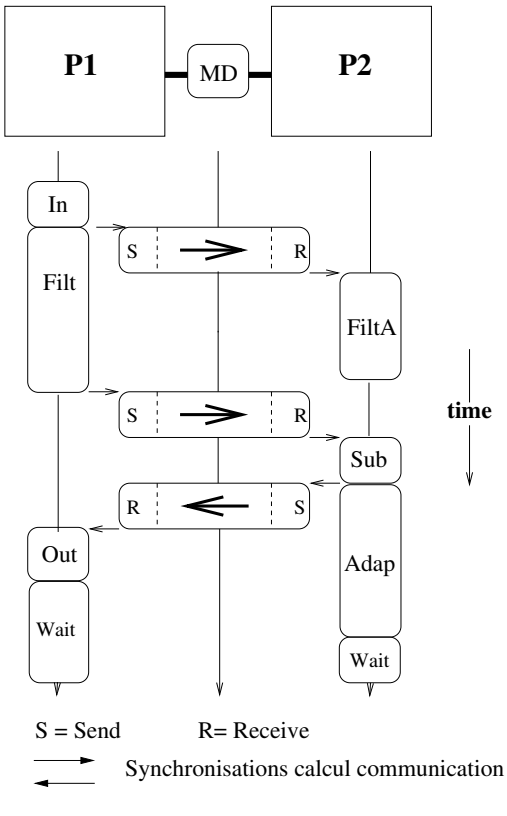
- **macros système** constituant le **macro-exécutif** :
  - **conditionnement** : choix des opérations en fonction d'une condition,
  - **attente (`wait`)** : délai à la fin d'une opération pour assurer sa période,
  - **communication** inter-processeur : pour les SAM envoi d'un message de donnée d'un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

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. De telles conceptions sont "insensibles aux latences" LID Latency Insensible Design.

Génération de code

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

Architecture TT comprenant 2 processeurs `P1` et `P2`, avec chacun un opérateur et un communicateur, et un médium point-à-point.



Opérateur de P1 : macro-boucle de tâches, macros (`In`, `Filt`, `Out`, `Wait`).
Communicateur de P1 : macro-boucle de communications, macros (`send`, `send` et `recv`).
Opérateur de P2 : macro-boucle de tâches, macros (`FiltA`, `Sub`, `Adap`, `Wait`).
Communicateur de P2 : macro-boucle de communications, macros (`recv`, `recv`, `send`).
Chaque boucle de tâches d'un opérateur et chaque boucle de communications d'un communicateur contiennent des macros supplémentaires de synchronisation entre opérateur et communicateur.
Les macros **wait** assurent un auto-déclenchement à période donnée, supérieure à la latence optimisée.

Génération 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** d'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 de code

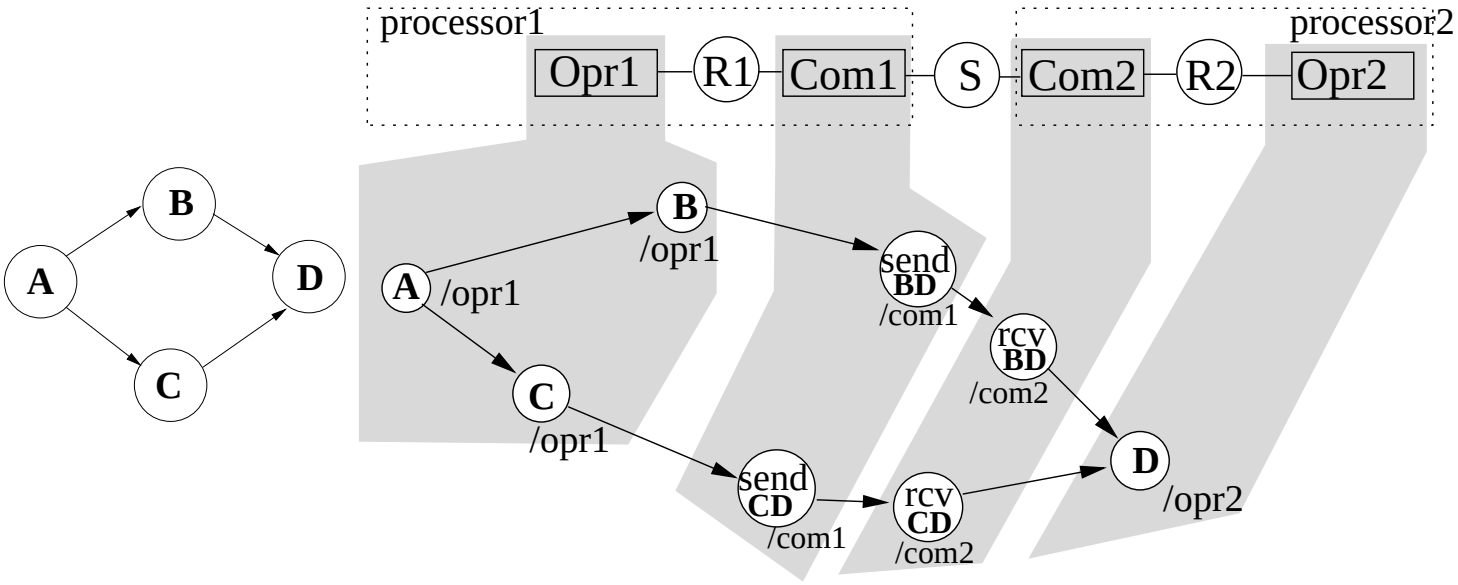
Synchronisations dans le macro-code : principes 2/2

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

- pour le passage de messages :
  - médium point-à-point : pas de messages de synchronisation car la FIFO est synchronisante,
  - 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 de code

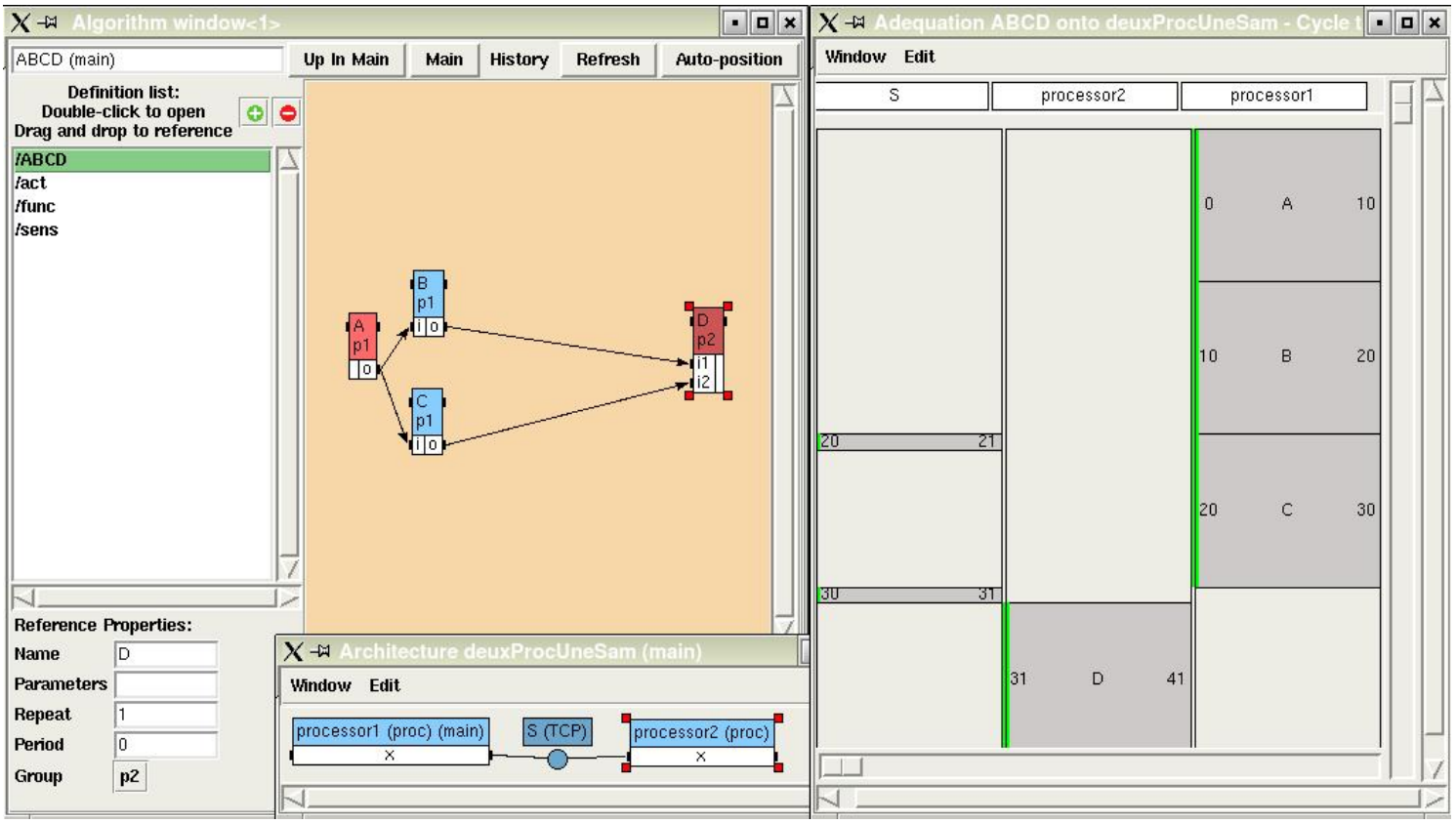
Synchronisations : Graphes d'algorithme ABCD, d'architecture, d'implantation



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

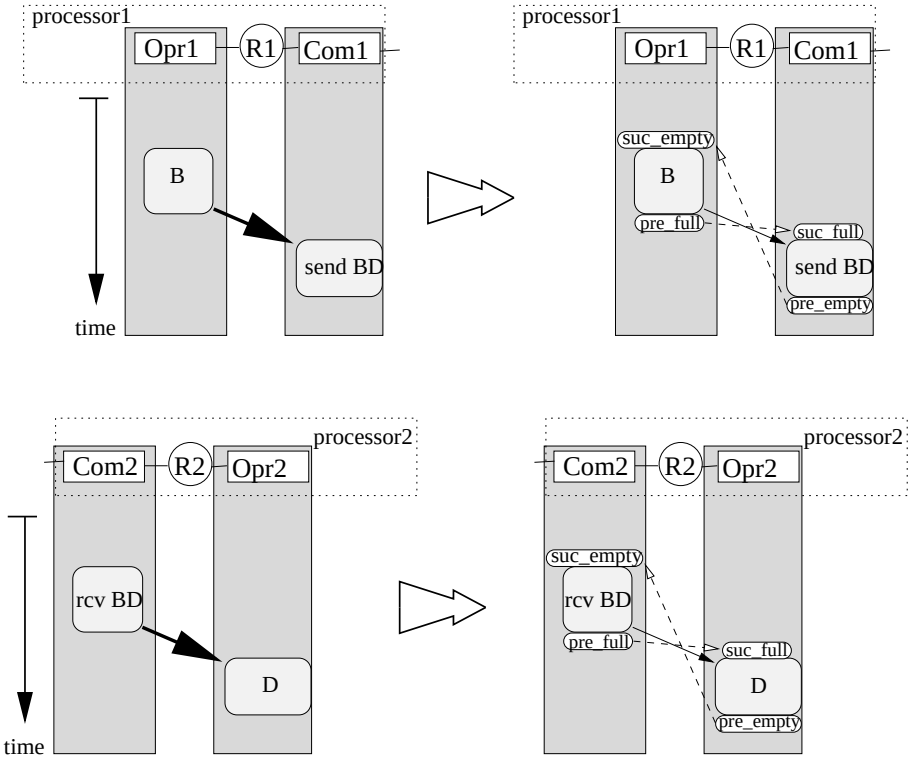
Génération de code

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



Génération 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`.

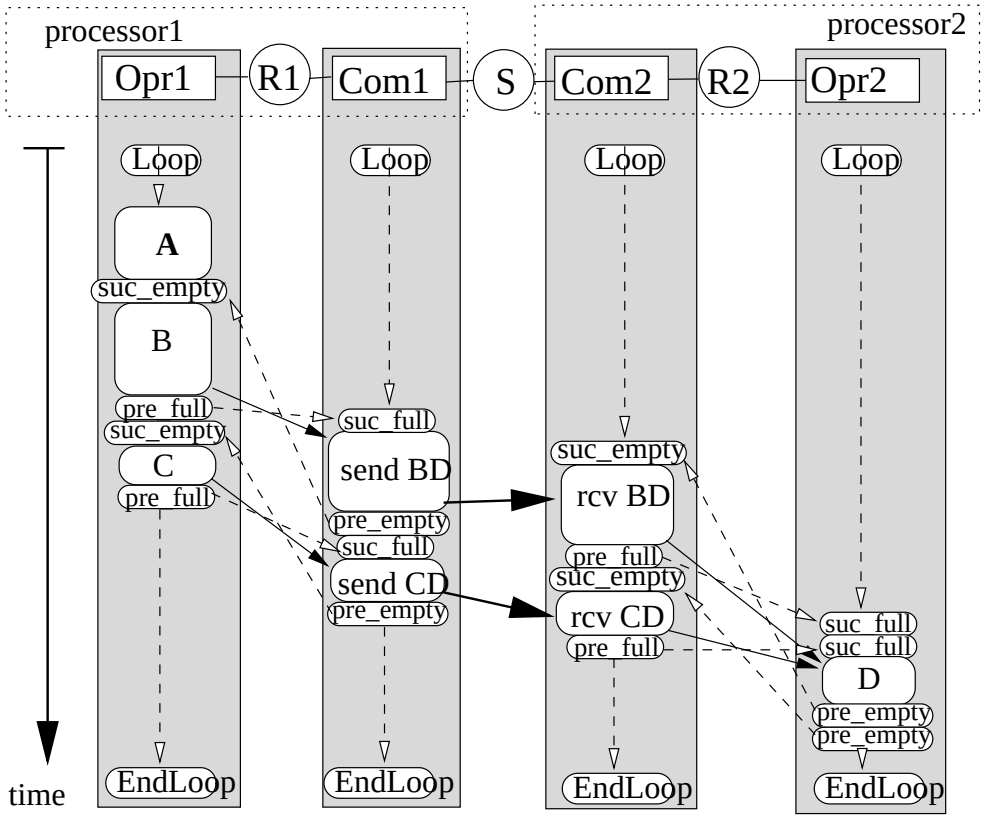
Synchronisations inter-répétition (Suc_empty, Pre_empty) : tampon vide avant envoi donnée.

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

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

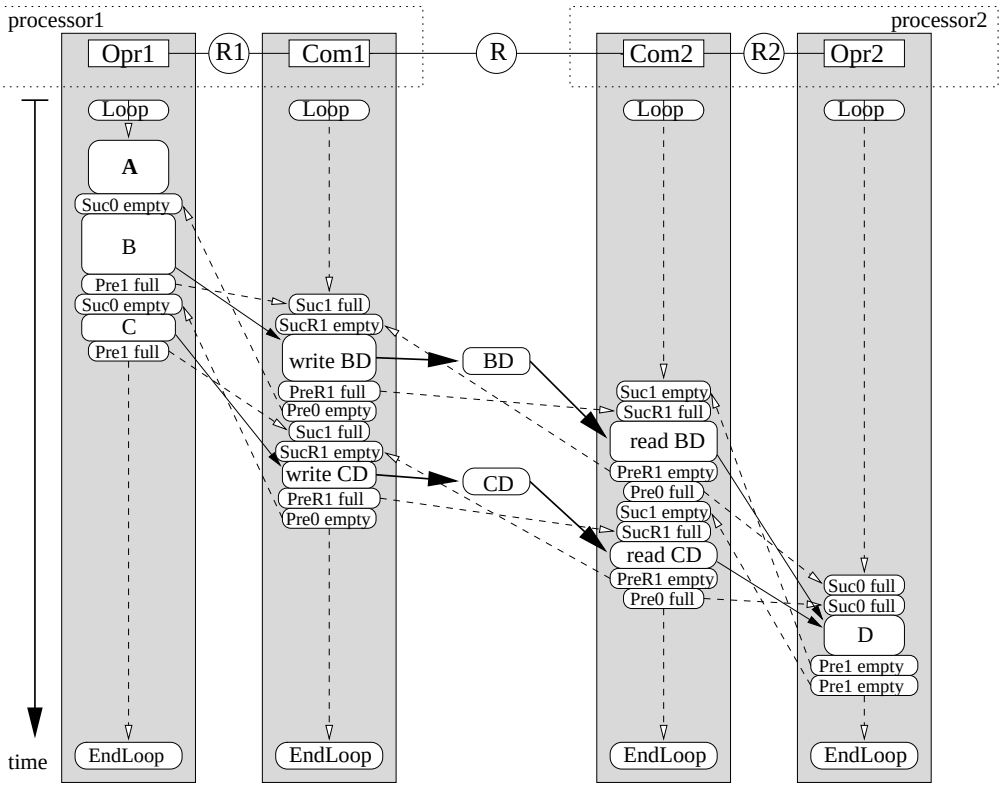
Génération de code

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



Génération de code

Synchronisations intra-processeur : communication par mémoire partagée, algo ABCD entier



Génération de code

Génération de macro-code : synchronisation intra-processeur communication par passage de messages point-à-point, algo 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)
```

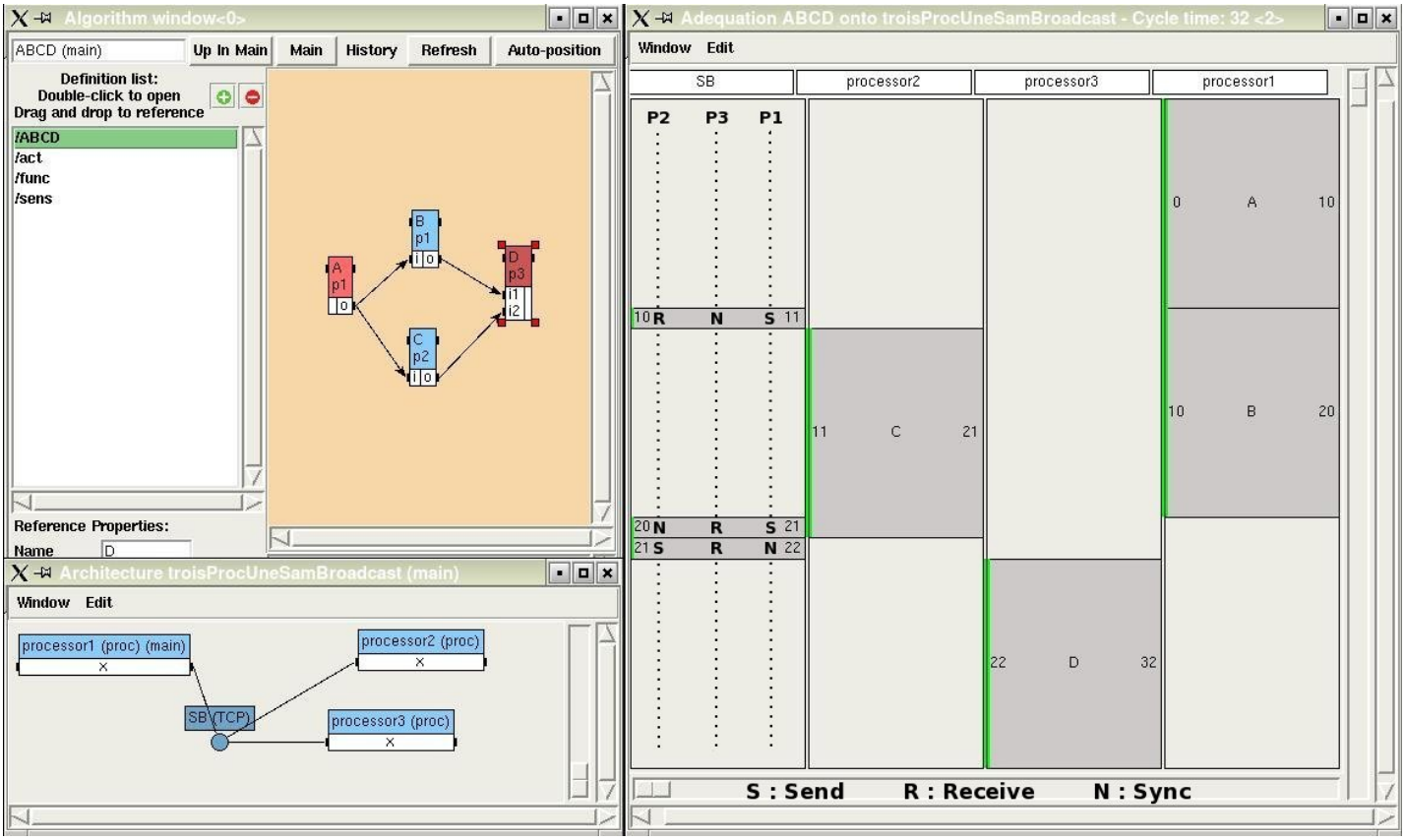
```
main_
  spawn_thread_(x)
  sens(_ABCD_A_o)
  loop_
    sens(_ABCD_A_o)
    Suc0_(_ABCD_B_o_processor1_x_empty,x,_ABCD_B_o,empty)
    func(_ABCD_A_o,_ABCD_B_o)
    Pre1_(_ABCD_B_o_processor1_x_full,x,_ABCD_B_o,full)
    Suc0_(_ABCD_C_o_processor1_x_empty,x,_ABCD_C_o,empty)
    func(_ABCD_A_o,_ABCD_C_o)
    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_
```

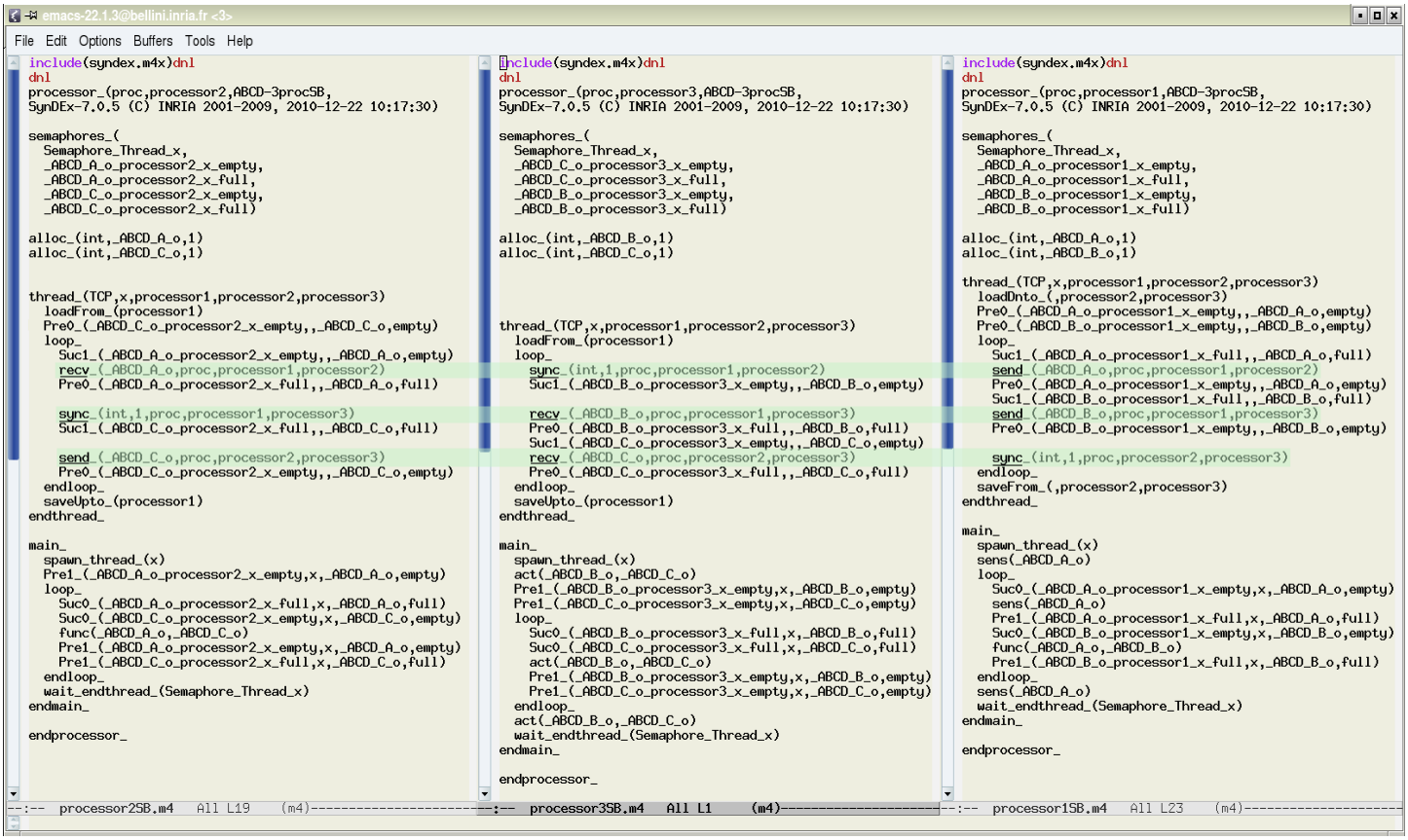
Génération de code

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



Génération de code

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



Génération de code

Génération de code exécutable distribué temps réel embarqué 1/2

Pour chaque processeur son macro-code est macro-processé avec :

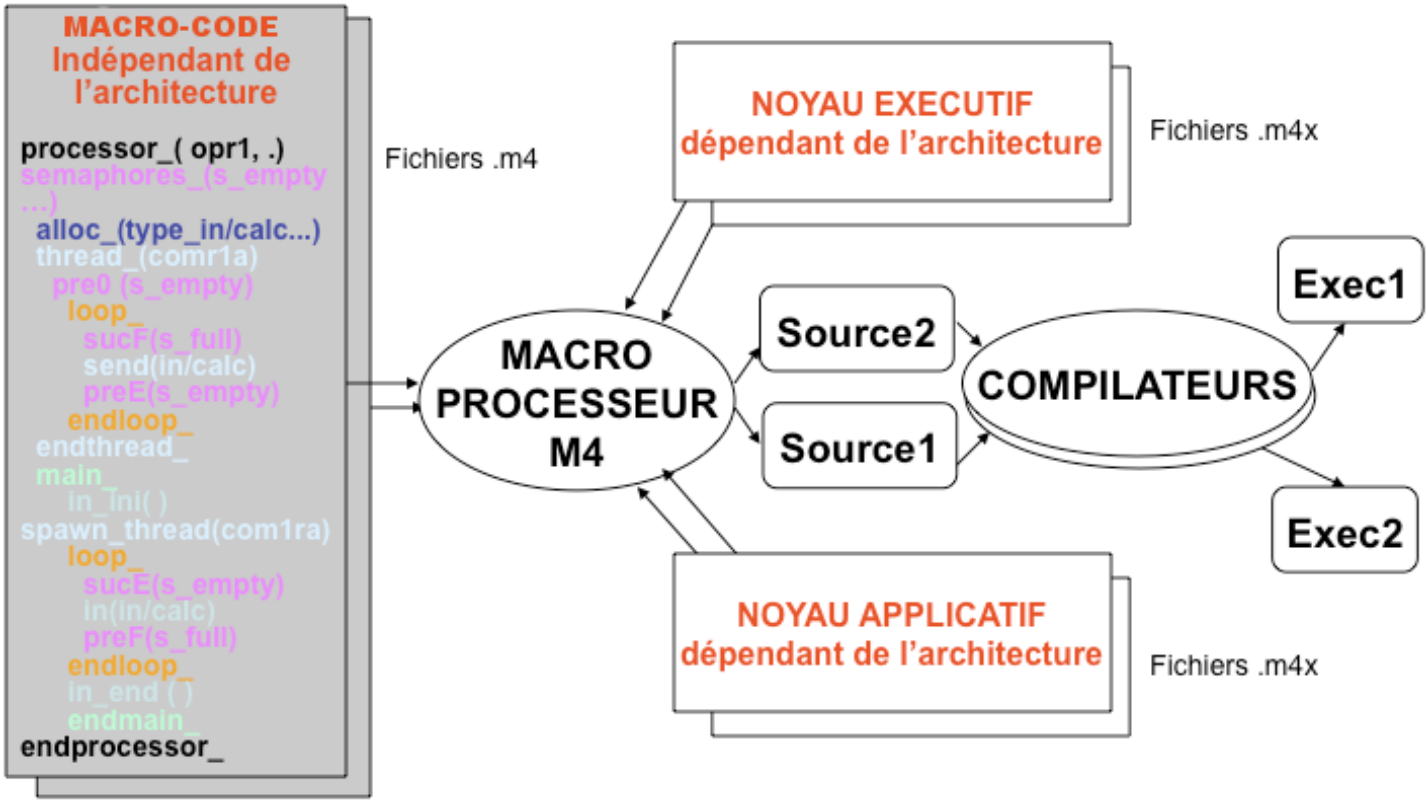
- un **noyau d'exécutif dépendant de l'architecture** et de l'éventuel **exécutif résident**, par exemple VxWorks, Osek, Linux/RTAI, Windows/RTX, etc. Chaque noyau d'exécutif contient les informations décrivant comment chaque macro-instruction sera traduite en du code source compilable;
- un **noyau applicatif dépendant de l'architecture** contenant les macro-opérations correspondant aux opérations du graphe d'algorithme. Chaque noyau applicatif contient les informations décrivant comment chaque macro-opération sera traduite en du code source compilable.

Les sources obtenus seront compilés pour produire les exécutables.

Pour 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 de code

Génération de code exécutable distribué temps réel embarqué 2/2



Logiciel SynDEx

Fonctionnalités 1/2

Mise en œuvre de la méthodologie AAA

SynDEx est un logiciel graphique interactif 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 offre les fonctionnalités suivantes :

- spécification fonctionnelle,
  - spécification du graphe d'algorithme avec un langage propriétaire,
  - 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 pour le temps réel embarqué), etc., générant ce langage propriétaire,
- spécification extra-fonctionnelle,
  - spécification du graphe multicomposant,
  - caratérisations,

Logiciel SynDEx

Fonctionnalités 2/2

- adéquation,
  - analyse d'ordonnançabilité temps réel distribuée,
  - 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, 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.

Logiciel SynDEx

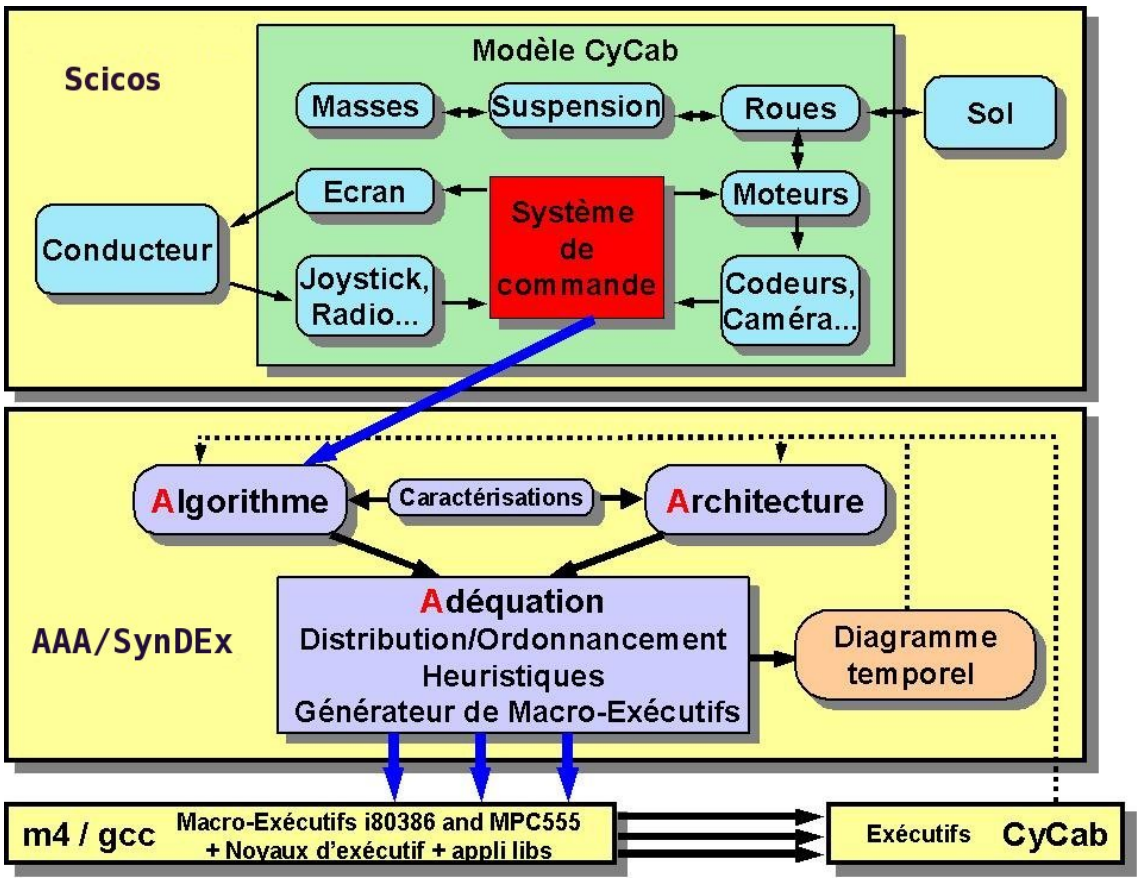
Exemple CyCab 1/2

- 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

Logiciel SynDEx

Exemple CyCab 2/2



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 calcule les dates d'exécution de chaque opération de calcul et de communication à partir des périodes, des durées d'exécution des opérations de calcul et des opérations de communication ajoutées. 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.

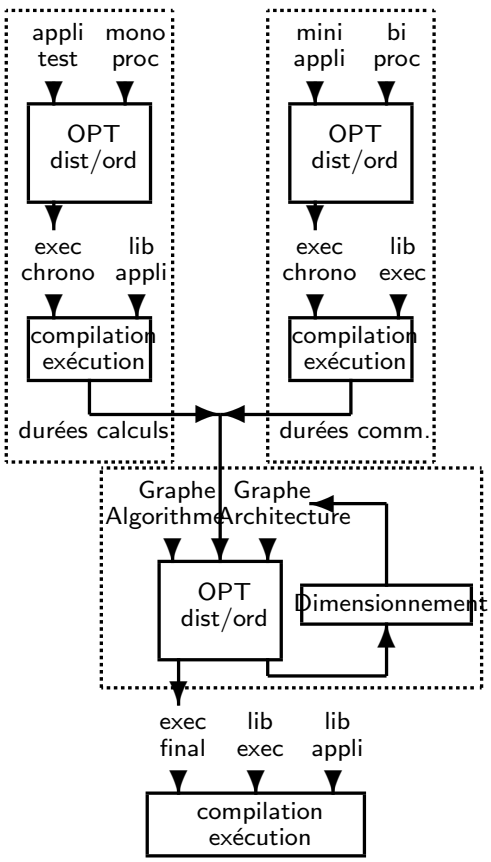
Logiciel SynDEx

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 chronomètre donnant la date avant l'exécution de l'opération et un second chronomètre donnant la date après exécution.

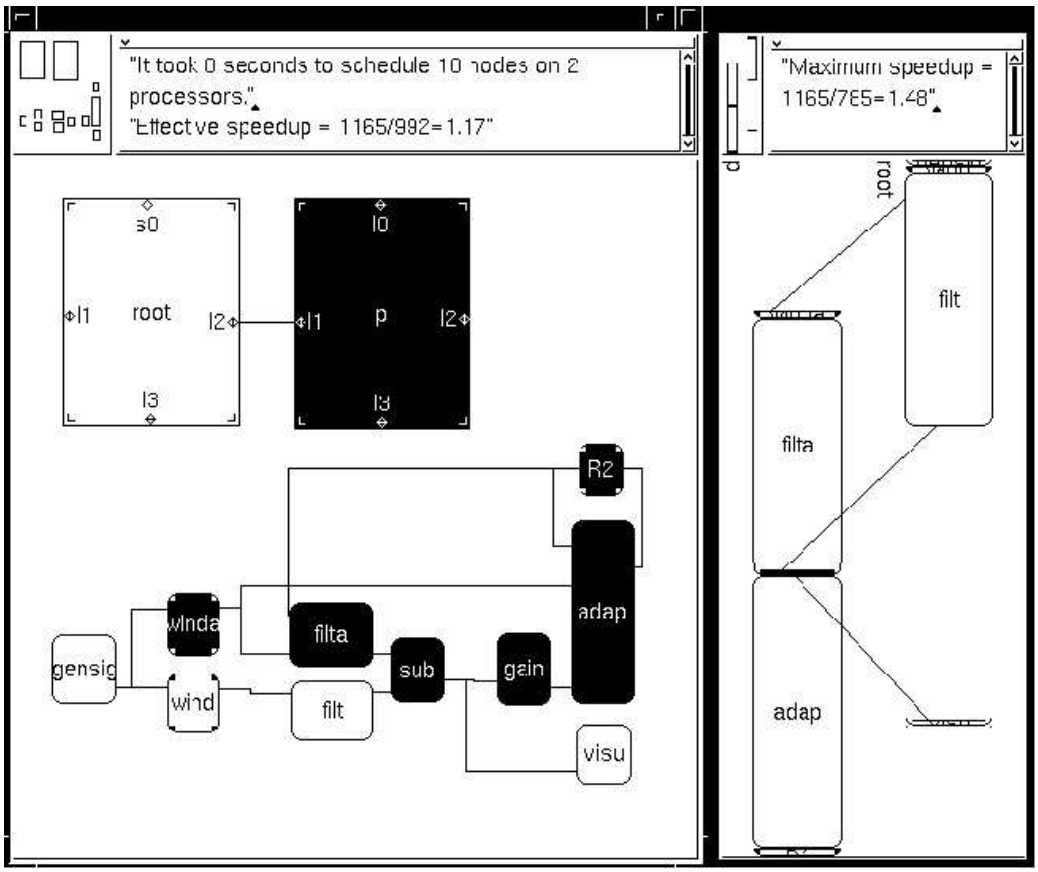
L'application considérée est exécutée en monoprocesseur et la différence entre les dates avant-après donne la durée d'exécution de chacune de ses opérations.

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



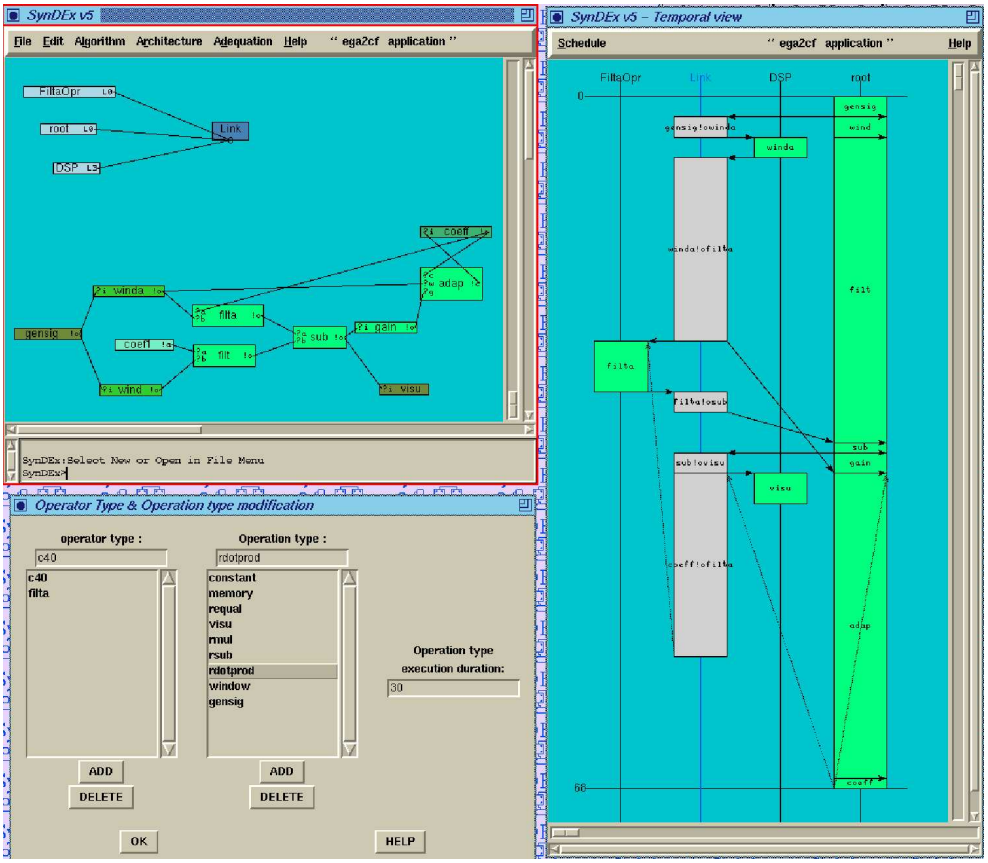
Logiciel SynDEx

IHM V4



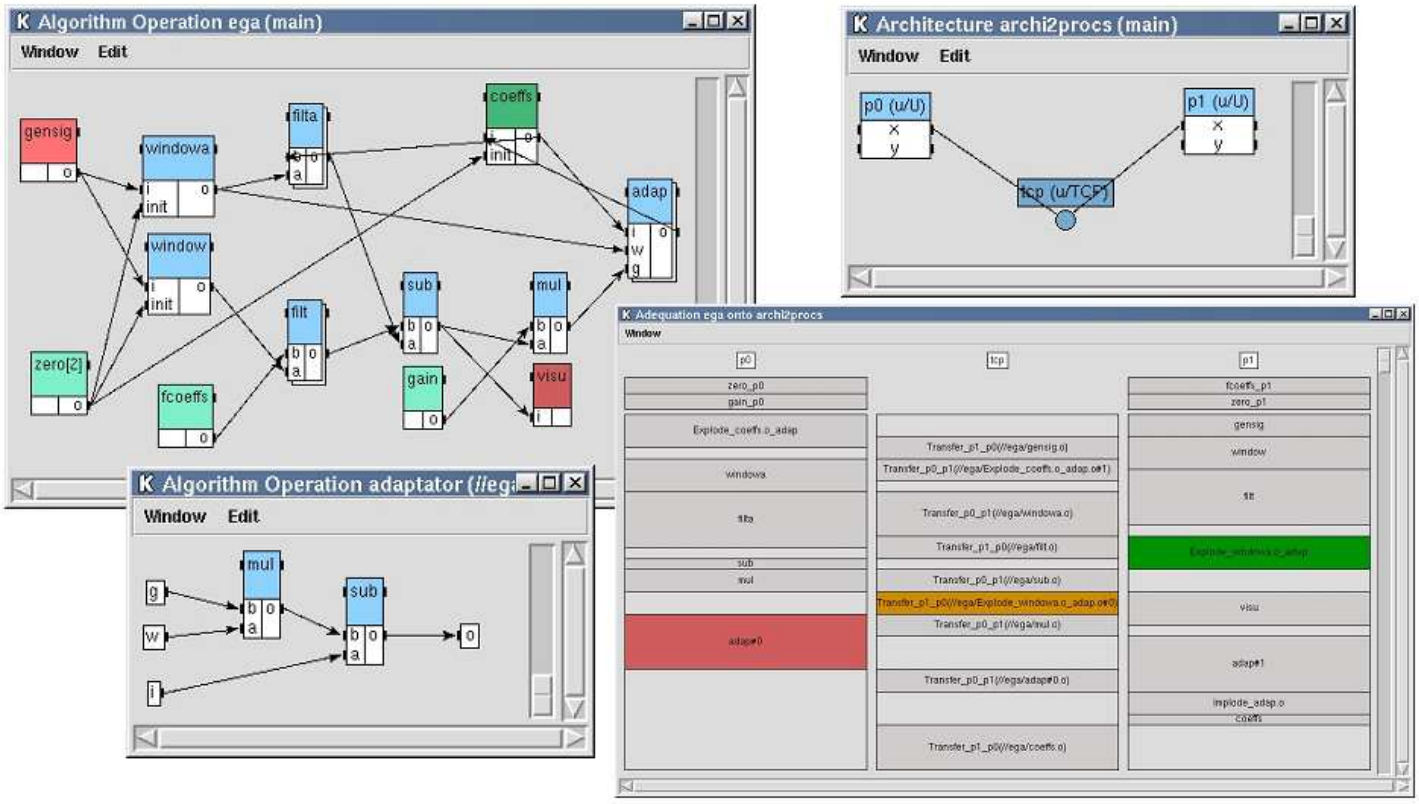
Logiciel SynDEx

IHM V5



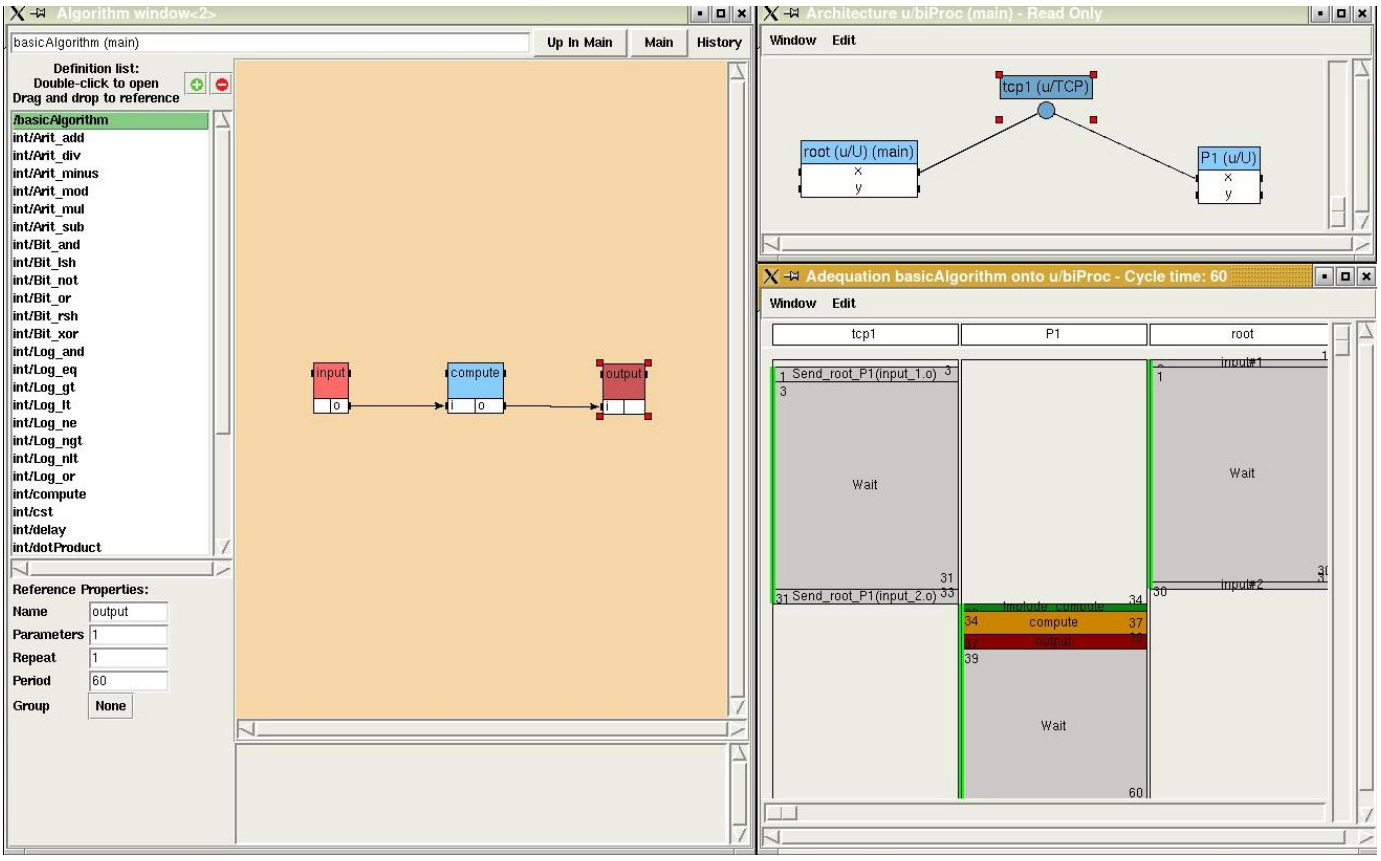
Logiciel SynDEx

IHM V6



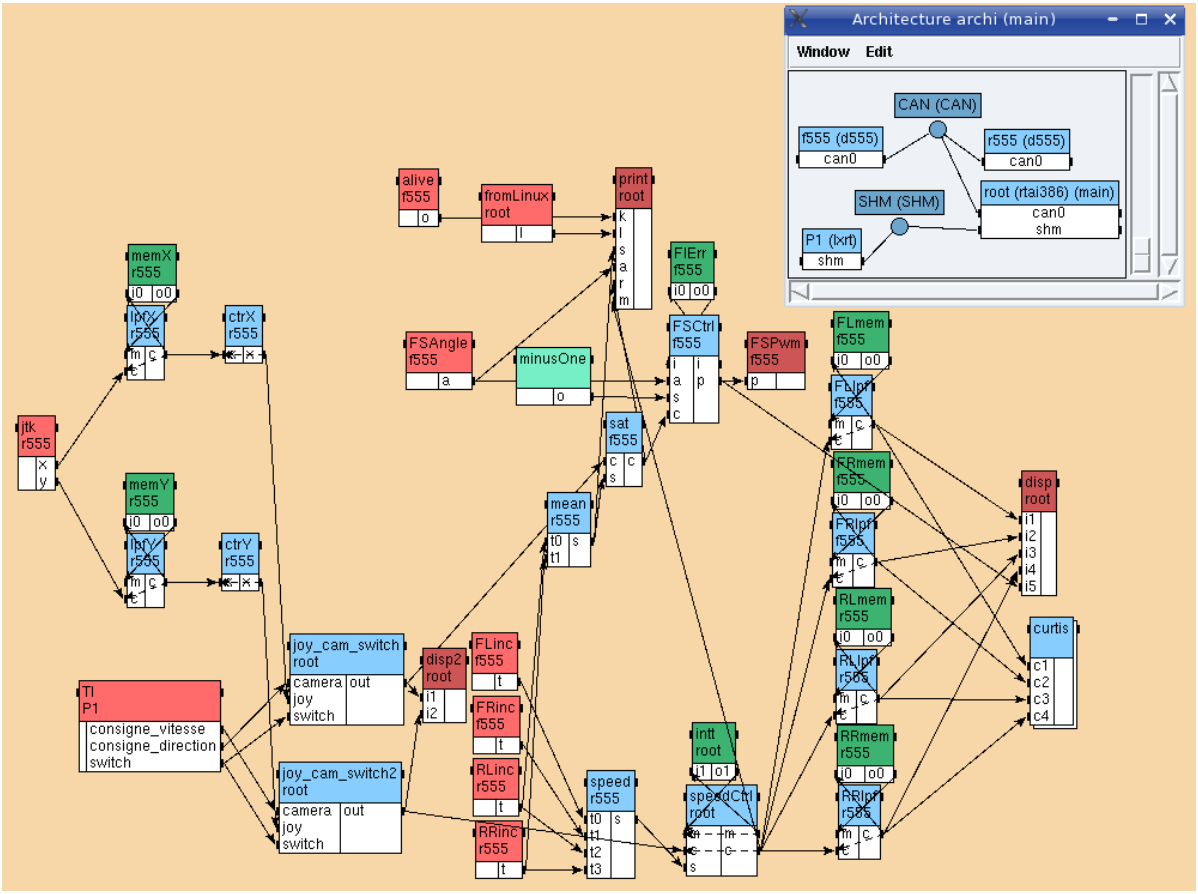
Implantation optimisée

IHM V7 (Algo simple : input 30ms, compute et output 60ms)



Logiciel SynDEx

IHM V7 (Suivi CyCab : TI 100ms, ctrl-x 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 extra-fonctionnelle permet de décrire les **ressources matérielles** et les **caractéristiques temporelles**.
- La méthodologie AAA permet de faire des spécifications fonctionnelles et extra-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 logiciel SynDEx **concrétise** la méthodologie AAA.

Conclusion

Conception sûre par construction

Implantation optimisée : adéquation

Temps de développement réduit