# Master of Science

## A Formal Approach for Safe Optimized Distributed Real-Time Systems

### The Algorithm-Architecture Adequation (AAA) Methodology

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

# Plan I



# Plan II

# Context and goals

## Embedded systems examples

Automotive

© Renault, Crédits D.R

Avionics


© Airbus S.A.S 2005

Mobile robotics

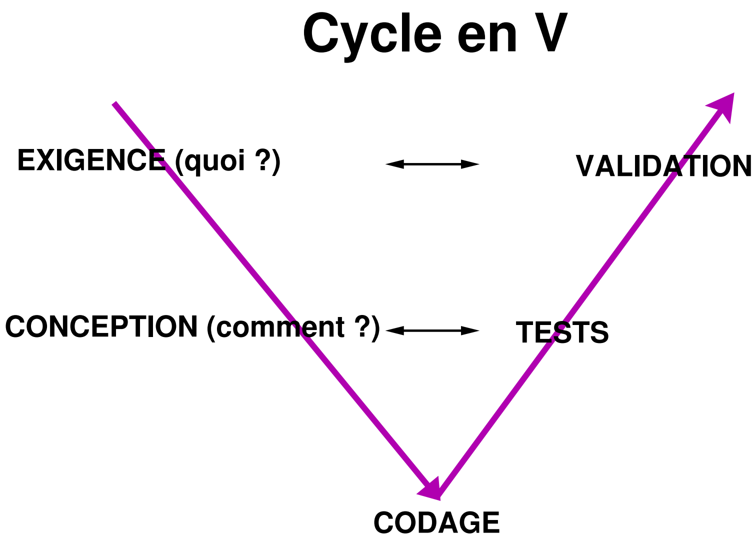

© INRIA / Photo J.-M. Ramès INRIA

Telecommunication

© Raphaël Soret / France Télécom

# System approach

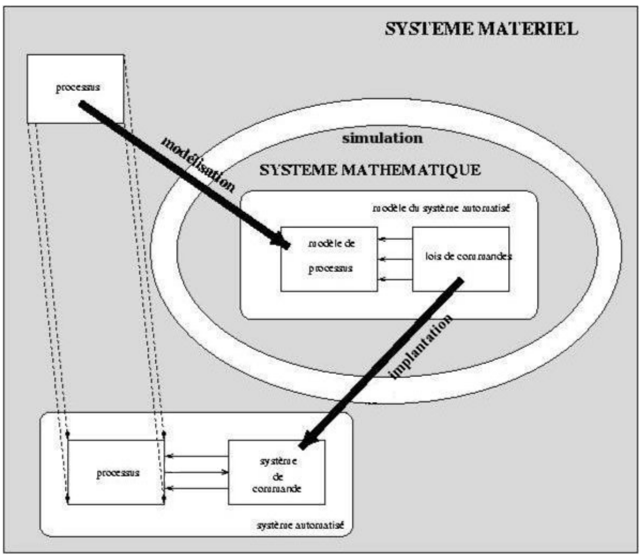
## Development life cycle of embedded systems




We focus on the lower part of the **V** development life cycle: **Design** $\leftrightarrow$ **Tests** and **Coding**. The main goal is to automate the coding and tests or at least minimize them. We aim a **I** (down-side of the cycle) enabling a **safe by construction design**, thus avoiding the up-side of the cycle.

# System approach

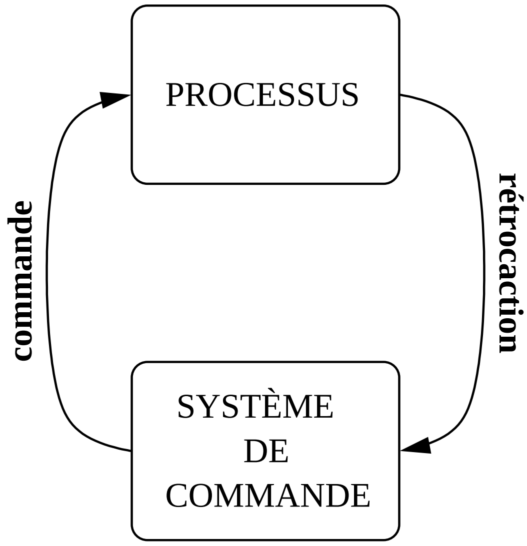
## Links between CONTROL THEORY and COMPUTER SCIENCE to design a system



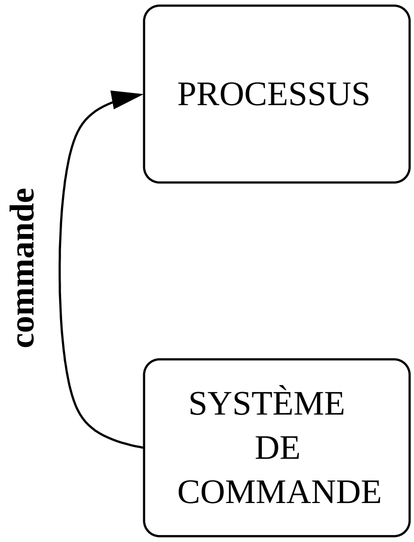

**CONTROL THEORY** - modelling/simulation - **Specification**
Modeling/hybrid simulation ot the system: process in continuous and/or discrete domains, control system in discrete domain.
**COMPUTER SCIENCE** - implementation - **Design Coding**
Control system implementation on processors and/or on specific integrated cirtuits (IC), then connection with the process.

# System approach

## General structure of the system



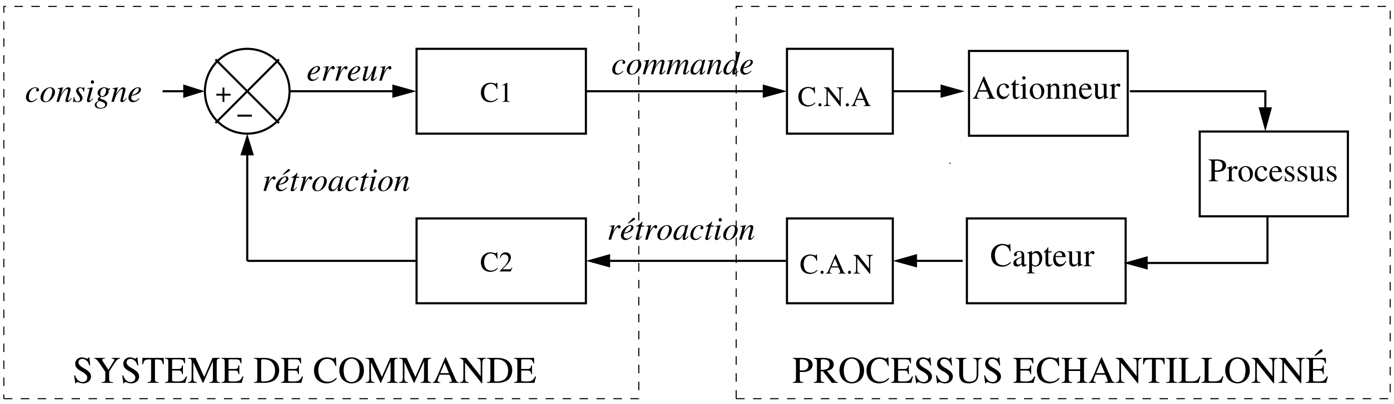

(a) closed loop




(b) open loop

# System approach

## Control system and sampled process loop



# Definitions

## Reactive system

| | | | | |
|---|---|---|---|---|
| Input events<br>Stimulus | $\Rightarrow$ | CONTROL SYSTEM<br>REACTIVE AND REAL-TIME | $\Rightarrow$ | Output events<br>Reaction |

**Reactive system** (Harel, Pnueli 1985): the control system whenever it consumes an **input event**, executes functions and **must** produce an **output event**.

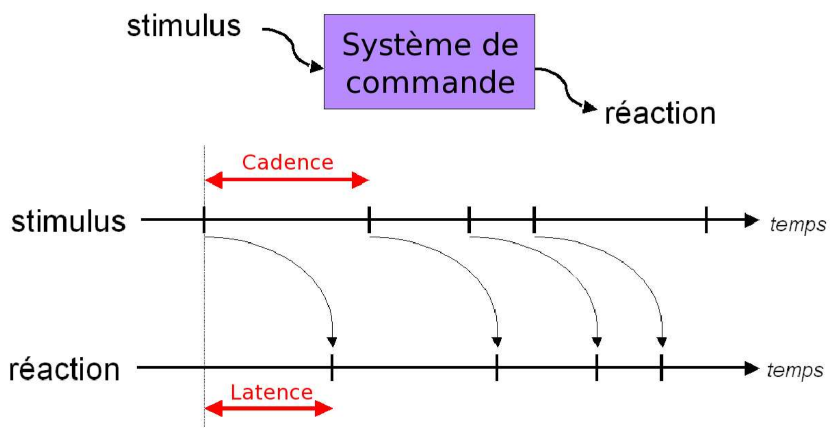
The control system consumes an infinity of input events, numerical values produced by the process trough a **sensor**, associated to an analog digital converter (ADC). The control system produces an infinity of ouput events, numerical values consumed by the process through an **actuator** associated, to a digital analog converter (DAC). Every infinite sequence of events is called a **signal**.

# Definitions

## Real-time system

**Real-time system**: reactive system that must satisfy constraints of two types:

- **input rate**: constraint on the duration between the occurence of two successives event of a signal (period, sporadic with a minimal period, aperiodic without period),
- **input-output latency**: constraint on the duration of the reaction triggered by an event of an input signal and producing an event of an output signal.



# Definitions

## Distributed embedded real-time system, event or time triggered

**Hard, critical, strict real-time system**: all constraints must be satisfied otherwise catastrophic consequences occur: human beings loss, ecological disaster, etc.

**Soft, QoS, real-time system**: some percentage of constraints may not be satified, quality of service.

**Distributed, parallel, multiprocessor, multicore system**: for performance, modularity, bring closer computation and sensors/actuators.

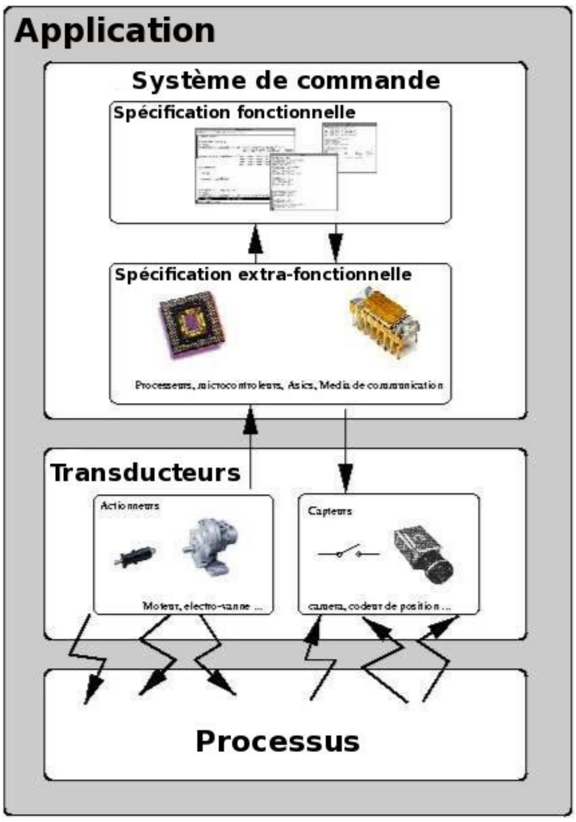
**Embedded system**: require resource minimizations (volume, weight, power consumption, cost, etc.).

**Event triggered system (ET)**: the process state is known through interruptions provided by sensors, actuators must follow sensor rate, flexible, probabilistic prediction, suited to soft real-time.

**Time triggered system (TT)**: the process state is known through periodic polling of sensors and actuators relatively to a discrete time (quantum), actuators must be synchronized, not flexible, deterministic prediction, suited to hard real-time.

# Definitions

## Real-time application



# Application domains

## Consumer and large scale systems

- **Consumer product systems**
  - telecoms: smart mobile phone, adsl modems, etc.
  - automotive electronics: engine control, driver assistance, etc.
  - robotics: automatic vehicle, cleaner, industrial robot, etc.
  - medical electronics: implants, patient monitoring, etc.
  - domotic: remote monitoring, automatic vacuum cleaner, etc.
  - audio and video equipements: walkman, set-up box, HD television, etc.

**Goal: cost minimization**

- **Large scale systems**
  - aeronautics and spatial
  - air-trafic control
  - railroad system
  - industrial control process of plant
  - telecomunication infrastructure
  - weapon system

**Goal: development cycle minimization**

# Functional specification

## Image and signal processing - control

Specification of functions and of **data dependences** relating output of functions **producing** data and input of functions **consuming** data.

**Control theory**

| | | |
|---|---|---|
| **Signal image processing** | $\Rightarrow$ | Numerous computations<br>Regular - `For i=1 to N Do` |
| **Control** | $\Rightarrow$ | Few computations<br>Non regular - `If cond Then Else`<br>Mode changes |

Generally, regular and non regular algorithms are mixed, increasing the complexity of the implementation problem.

# Non functional specification

## Hardware architecture

Specification of the hardware architecture components and their interconnections. Specification of distribution and scheduling constraints on the functions relatively to hardware components and on data dependences relatively to communication devices.

$$\frac{\text{Function execution time}}{\text{Latency constraint}} > 1 \Rightarrow \text{Distributed, parallel, etc., architecture}$$

Heterogeneous architecture called **multicomponent** (Lavarenne, Sorel 96) composed of:

- **sensors** and **actuators**,
- **programmable components**: processors RISC, CISC, DSP (Digital Signal Processor), ASIP (Application Specific Instruction set Processor),
- **non programmable components**: specific electronic board, ASIC (Application Specific Integrated Circuit), FPGA (Field Programmable Gate Array),
- **communication medium**: point-to-point link (crossbar), multi-point link (bus), network, etc.

# Non functional specification

## Timing caracteristics

Specification of timing characteristics associated to every function. They are of two types:

- **architecture independent**: period, minimal period, deadline constraints, generalized latency constraints on pair of functions not necessarily input-output,
- **architecture dependent**: Worst Case Execution Time (WCET) of functions on computational components and Worst Case Communication Time (WCCT) of data dependences on communication media.

Specification of **safety** and **security** properties **will not be considered in the course**.

# Optimized implementation: AAA methodology

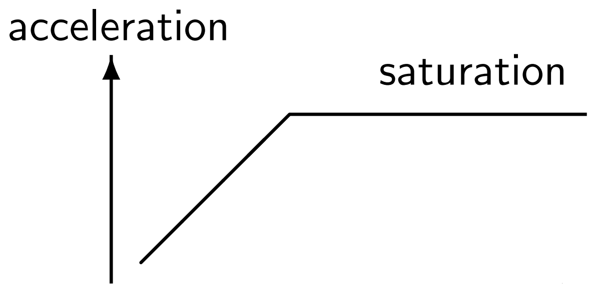
## Potential parallelism potentiel, actual parallelism

The **potential parallelism** (concurrence) of the functional specification is defined from the set of functions that are not dependent, indeed these functions will be executed potentially in parallel according to the **actual parallelism** of the architecture.

The **implementation** consists in choosing on which processor of the architecture each function will be distributed and scheduled.

When the actual parallelism of the architecture is less or equal to the potential parallelism, some acceleration may occur relatively to the uniprocessor execution time, this acceleration is proportional to the number of processors. As soon as the potential parallelism is greater, the acceleration does not increase any more, leading to a saturation phenomenom.

$$\text{Acceleration} = \frac{T_{uniprocessor}}{T_{multiprocessor}}$$




# Optimized implementation: AAA methodology

## Operating system role

Algorithm

Operating system

Architecture

**Algorithm** : informal definition (Al-Khwarizmi, astronomer Persia 825) description of a function using a finite number of instructions chosen in a finite set of instructions, more generic than a program which assumes that a language was chosen, formal definition (Turing, Post 1930).

**Operating System (OS)**: provides services to functions in order to take advantage of the architecture: programs, peripheral devices, memory, communications, synchronizations, possibly depending of time RTOS (Real-Time OS) also called **real-time executive** or **executive** later on.

**Architecture**: digital electronics composed of processors and/or IC, all interconnected.

# Optimized implementation: AAA methodology

## OS, RTOS, executive: distributed, reactive, real-time

**OS features**

| | | |
|---|---|---|
| OS | $=$ | Hardware resource allocation: computation, memory, communication to the algorithm |
| $+$ | | |
| Distributed | $\rightarrow$ | Several resources for every type |
| $+$ | | |

**RTOS features**

| | | |
|---|---|---|
| Reactive | $\rightarrow$ | Reaction order $=$ stimuli order independently of the reaction time |
| $+$ | | |
| Real-time | $\rightarrow$ | Allocation conditioned by physical time flow |

# Optimized implementation: AAA methodology

## Real-time executive: resource allocation

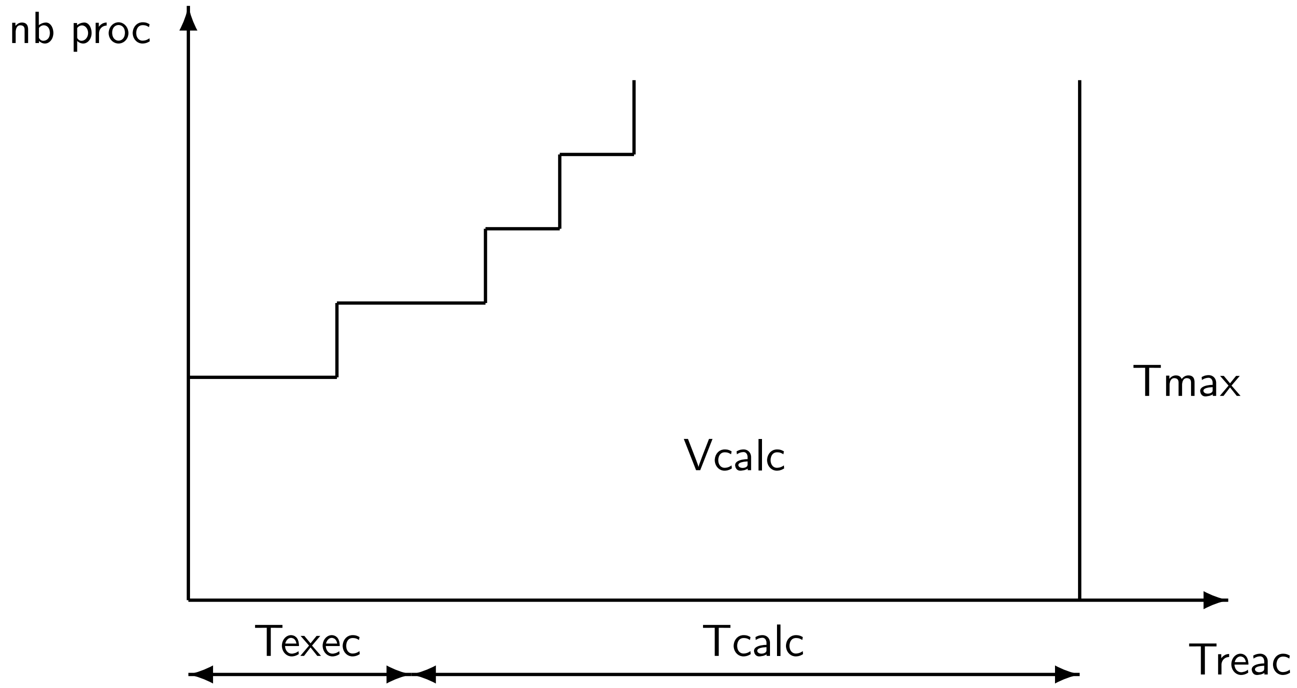
**The real-time executive allocates resources**

- Distribution: spatial allocation
- Scheduling: timing allocation
- Off-line: optimizations and decisions performed **before** execution (execution duration must be known)
- On-line: optimizations and decisions performed **during** execution (use of real-time clocks)

# Optimized implementation: AAA methodology

## Real-time executive: executive overhead

**The real-time executive involves an overhead** which increases according to the number of processors.




# Optimized implementation: AAA methodology

## Goal

From a **functional specification** and a **non functional specification** (architecture and timing characteristics), explore all the possible implementations (spatial and timing allocation of functions to architecture resources, considered as sequencial machines) to obtain, manually or automatically an **optimized and safe by construction implementation**.

In order to reach this goal the exploration is achieved from **formal models** describing the algorithm and the architecture (graphs, partial order, finite state machines) by performing graph transformations based on **multiprocessor real-time schedulability analyses** and **timing and resources optimizations**.

# Optimized implementation: AAA methodology

## Basic models

A **graph** $G$ is a pair $(S, A)$ where $S$ is a finite set of vertices and $A$ is a binary relation on $S$ defining pairs of vertices $(s_1, s_2) \in S \times S$ such that $(s_1 \, A \, s_2) \Leftrightarrow (s_1$ "is related to" $s_2)$ through an **edge**. This graph is **directed** if every pair (edge) is ordered $(s_1, s_2) \neq (s_2, s_1)$, we have $s_1$ "precedes" $s_2$. An ordered pair is called a **directed edge**. This graph is **acyclic** if it does not have a sequence of directed edges such that $(s_1, s_2) \ldots (s_n, s_1)$.

For a **directed graph** the relation $A$ is antisymmetric, i.e. if $s_1 A s_2$ and $s_2 A s_1$ then $s_2 = s_1$, it is transitive and it is not reflexive. Thus, it is a **strict order relation** (noted $>$ different from $\geq$ that is non strict).

If the set of directed edges is such that $A \subset S \times S$ then $A$ is a **strict partial order relation**, $\bar{A} = S \times S$ is a **strict total order relation**.

A **stable** of the graph is a subset of vertices such that for every pair of vertices they are not in relation $A$, i.e. there is no edge connecting them, assuming this graph is not directed. The **greatest stable** of the graph is denoted by $A^*$, we have $A \cup A^* = \bar{A}$.

# Optimized implementation: AAA methodology

## Principles

| | **ADEQUATION** | |
|---|---|---|
| **ALGORITHM** | ← Timing → | **ARCHITECTURE** |
| ↓ | characteristics | ↓ |
| potential parallelism | ⟶ *reduction* | actual parallelism |
| ↓ | | ↓ |
| algorithm graph | ⟶ *transformations* | architecture graph |
| | **OPTIMIZATIONS** | |
| | ↙ ↘ | |

**Processors** | **specific IC**
**EXECUTIVE** | **NETLIST**

- custom synthetized
- may call a resident executive: VxWorks, Osek, Linux/RTAI, Linux/Xenomai, Windows/RTX, etc.

# Algorithm specification

## General issues

### Algorithm

The **functional specification** is performed by describing the control system as an **algorithm** that will be **implemented** on processors and/or specific integrated circuits all interconnected.

The algorithm describes, possibly hierarchically, the functions necessary to achieve the functional specification as well as the partial order of their execution, due to data dependences. It also describes the way some of these functions will be **conditionally executed** or **executed several times**.

There are **several approaches** to describe an algorithm using formal models based on graphs.

# Algorithm model

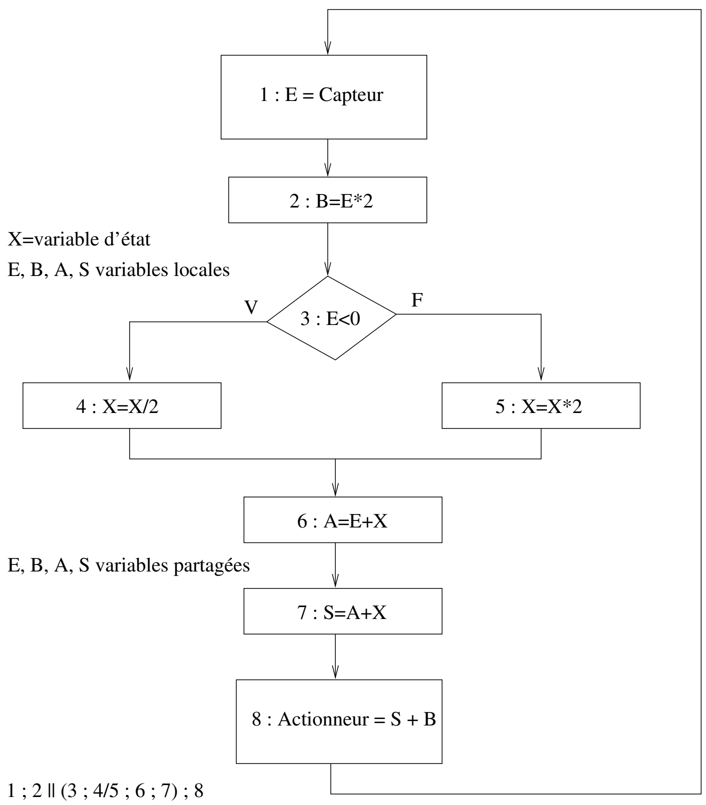
## Control flow graph 1/3

The algorithm can be modeled by a directed cyclic graph called a **control flow graph** of two possible types:

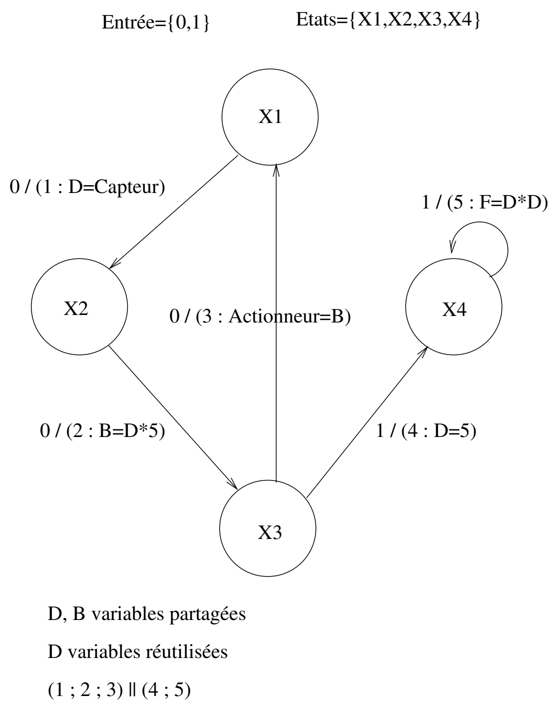
- a **flow chart** whose **vertices** are **operations** (functions) and **directed edges** are **control dependences** (foreward unconditional branching - sequential execution of operations - and test with foreward conditional branching for execution of alternative operations called `OR` divergence, loop with backward branching) which induce precedences between operations. An operation is executed as soon as the operation, it depends on, is completed, it reads and writes its data in local or state variables ;
- an **automaton** whose **vertices** are **states** and **directed edges** are **state transitions**. A transition is fired when an event occurs involving the execution of one operation which reads and writes its data in local variables ;
- interactions with the process (reactive system) are modeled through a loop (flow chart) and through event occurences (automaton).

# Algorithm model

## Control flow graph 2/3




ORGANIGRAMME




AUTOMATE

# Algorithm model

## Control flow graph 3/3

In this model:

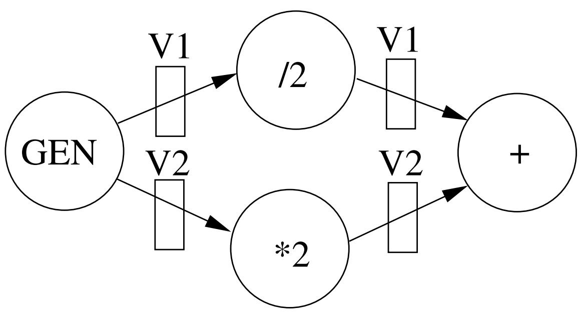
- operations access to the data memorized in variables which can be **reused and/or shared** by several operations, this mechanism is particularly **error-prone**.
- there is no relation between the order the data are accessed by operations and the order the operations are executed,
- all the operations are precedence related, the control flow induces a **total order** on the operation execution, no `AND` divergence.

In a flow chart model the state memory is implicitely localized in variables whereas in an automaton model it is explicitely localized in vertices.

# Algorithm model

## Data flow graph 1/3

The algorithm can be modeled by a directed acyclic graph (DAG) called **data flow graph** (Dennis, Kahn 1974) whose vertices are operations and directed edges are **data and control dependences**. A control dependence induces precedences between operations. An operation is executed as soon as all its input data - produced by operations that precede it - are available, thus it produces all its output data - consumed by operation that succeed it (Milner's activation rule). Vertices without predecessors are executed first. Similarly, vertices without successors are executed last.



# Algorithm model

## Data flow graph 2/3

In this model:

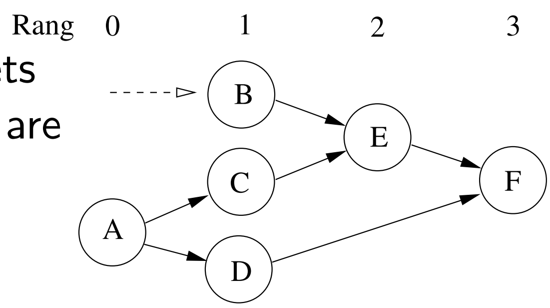
- to each directed edge corresponds a data transfer (single assignation) without shared variables and thus the corresponding errors, however the directed edges are implemented by variables that will be **automatically** managed by the compiler rather than the programmer,
- the order data are read ans written by operations is consistent with the order the operations, that uses that data, are executed, that order prevents the programmer to force an order on the operations and uses the variables in a different order leading to an error,
- some operations can not be precedence order related, the data and control flow induces a **partial order** on the execution of the operations: `AND` divergence.

# Algorithm model

## Data flow graph 3/3

- This partial order represents the **potential parallelism** inherent in the specification of the algorithm. More formally it is the **greatest stable of the graph** whose vertices in transitive relation are removed. In this set one determines all the vertices subsets that have the **same rank**, i.e. that are at the same distance, in terms of maximal number of predecessors, from the vertices without precedessors. It is possible to modify the rank of a vertex in order to increase the number of vertices of another rank.

  The greatest cardinal of these subsets gives the number of processors that are in **potential parallelism**, here $Card\{C, D, E\} = 3$;




- a vertex is **hierarchical** if it can be decomposed in another data flow graph, otherwise it is **atomic**. An atomic vertex cannot be allocated (distributed) on several resources of the architecture.

# Algorithm model

## AAA model: data flow graph repeated conditioned factorized 1/3

The proposed AAA model is a **data flow graph repeated conditioned factorized**, i.e. an extended data flow graph:

- **infinitely repeated**, every repetition of the data flow graph corresponds to an interaction between the control system described by the graph (reactive system), it defines a logical instant `t` of a **logical time** for a **LTT system** (Logical Time Triggered), vertices without predecessors correspond to **sensors** and vertices without successors to **actuators**,
- **conditioned**, i.e. a hierarchical vertex of the graph can be decomposed in several vertices such that only one of these vertices (`OR` divergence) will be executed every infinite repetition (logical instant) according to the value of its specific conditioning input, extension of the dynamic data flow graph (Buck 1993). Conditioning inputs are connected with a **conditioning dependance** to other operations. Conditioned vertices correspond to conditional branching in the control flow model (equivalent to `If...Then...Else...`);

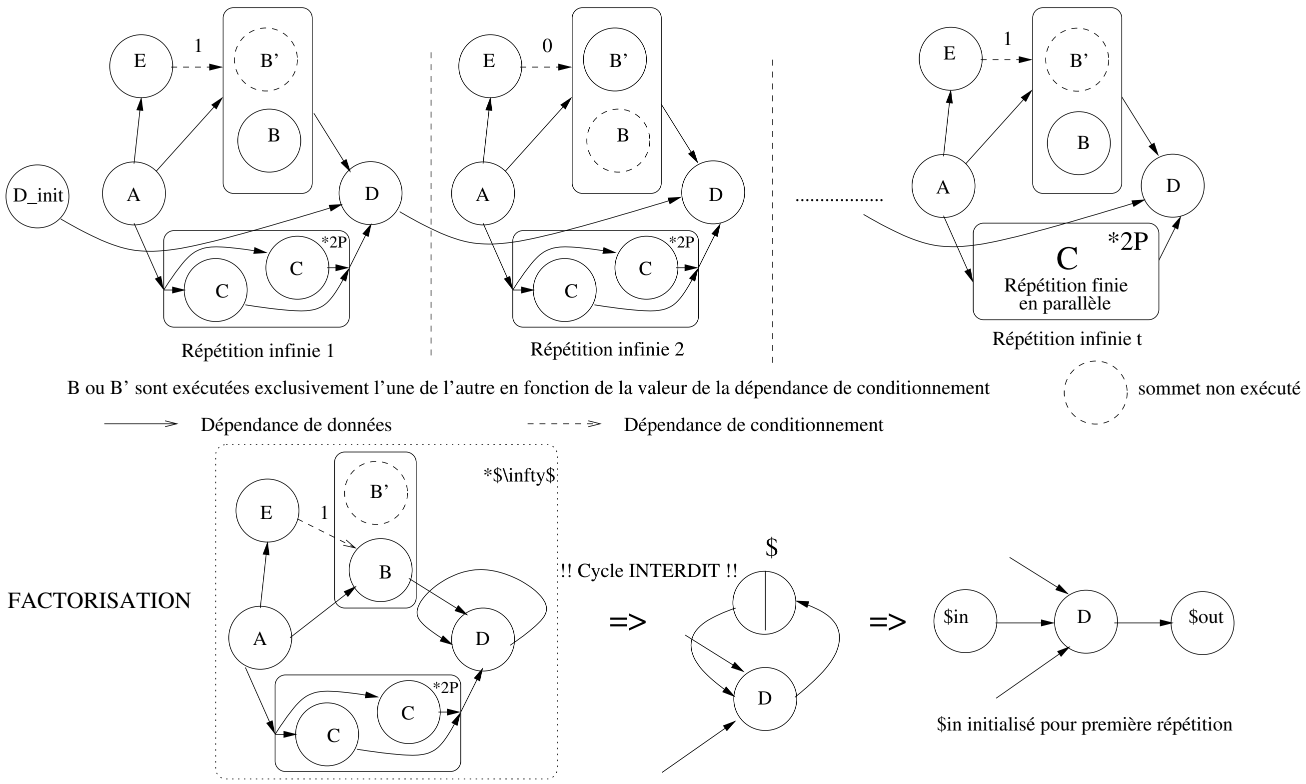
# Algorithm model

## AAA model: data flow graph repeated conditioned factorized 2/3

- **finitely repeated**, i.e. a hierarchical vertex of the graph can be decomposed in several identical vertices all executed on different data, corresponding to **potential data parallelism**, opposite to **potential control parallelism** where vertices are different, that is called by default **potential parallelism**. Finitely repeated vertices are represented as a single vertex with a repetition index called (P) for data parallelism and (S) for flow parallelism. It corresponds to loop in the control flow model (equivalent to `For...To...Do...`);
- **factorized** in order to simplify the model, but this may induce cycles when a repeated operation during infinite repetition $t$ consumes data produced during infinite repetition $t - n$. Cycles are **forbiden** to guarantee a deterministic behaviour, without deadlocks. Every cycle **must** contain at least a **delay vertice $** explicitely defining an **algorithm state**.

# Algorithm model

## AAA model: data flow graph repeated conditioned factorized 3/3

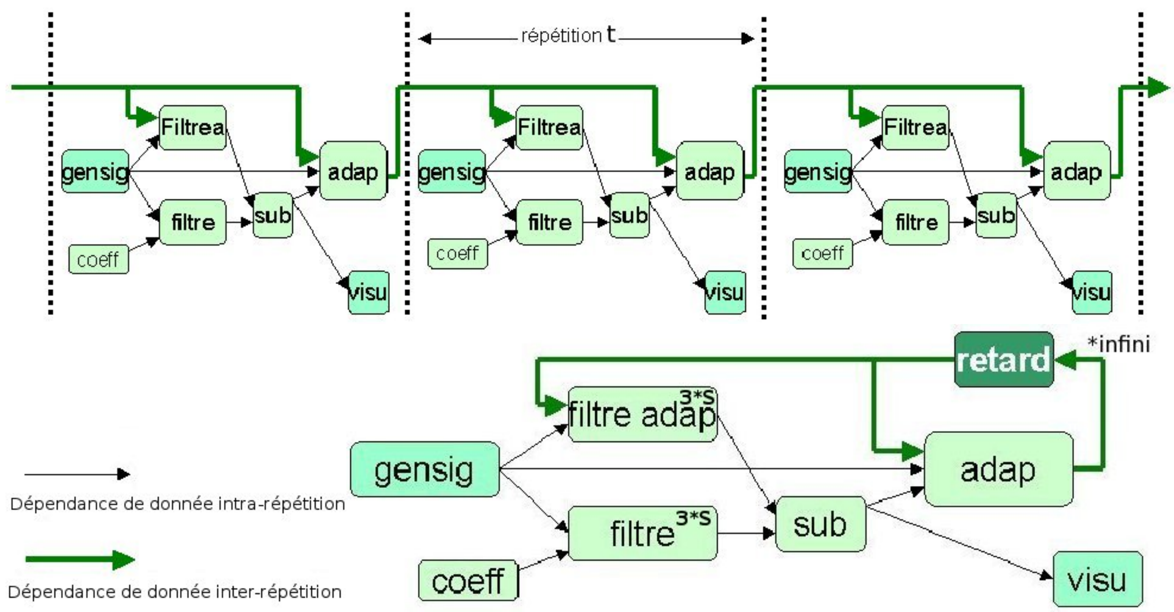



**Every infinite repetition corresponds to a logical instant $t$.**

# Algorithm model

## Example: adpative equalizer

The output of the sensor *gensig* is filtered by a FIR (Finite impulse response) digital filter with fixed coefficients and by another FIR whose coefficients are computed by an adaptive algorithm, both filter outputs are substracted and visualized by an actuator *visu*.

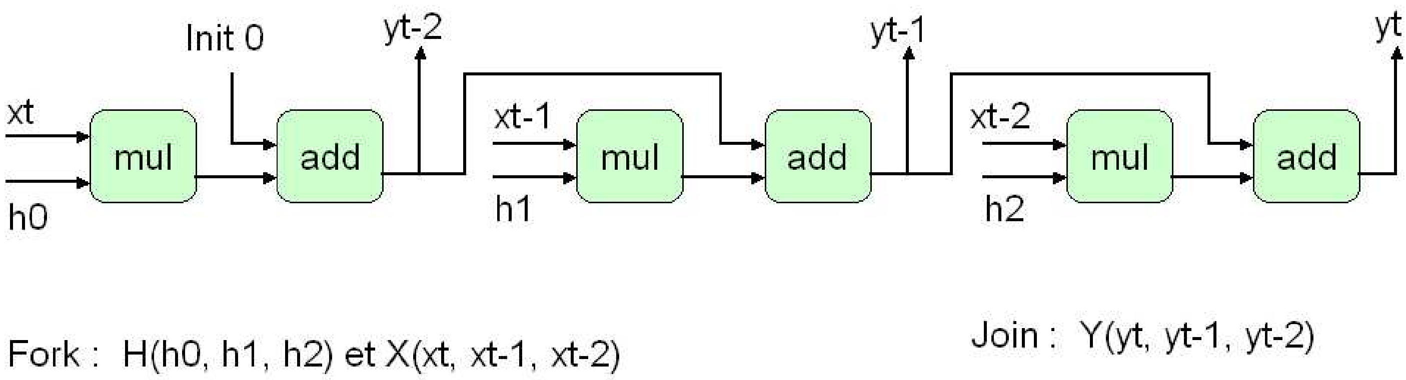


# Algorithm model

## Example: FIR filter

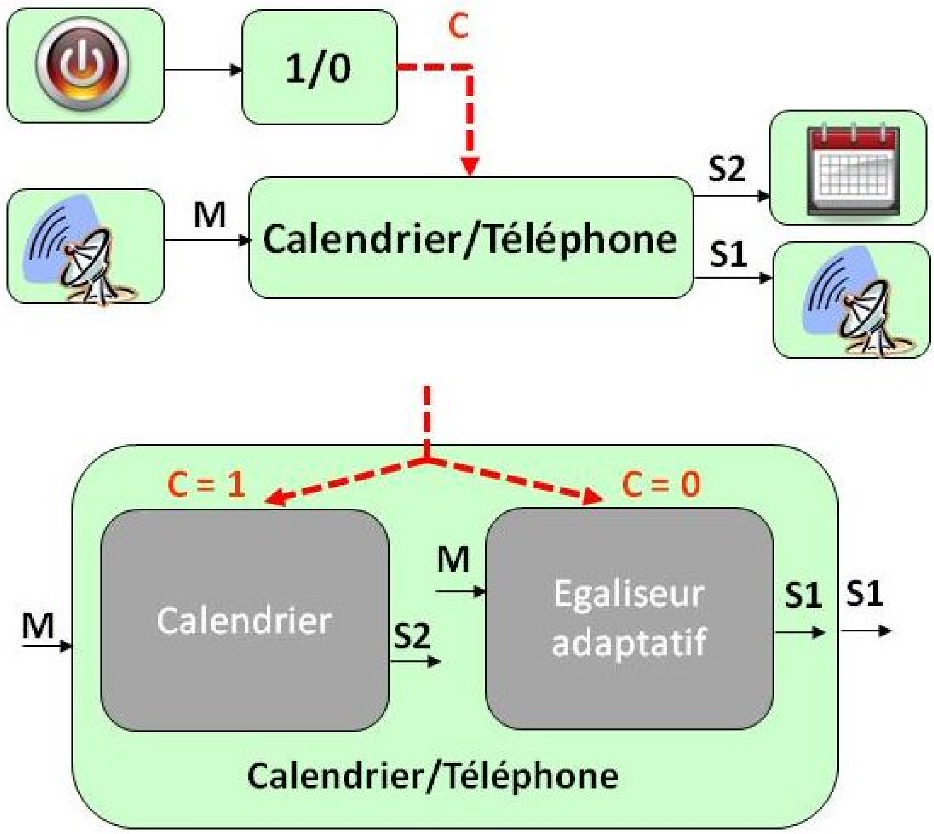
An input vector of 3 elements containing the coefficients $(h_0, h_1, h_2)$, an input vector of 3 elements containin the past values ot the input $(x_t, x_{t-1}, x_{t-2})$, a scalar output $Y_t = \sum_{i=0}^{2} h_i * x_{t-i}$

The graph (*mul*, *add*) is repeated 3 times (*3S) such that the output of every *add* is connected to the input of the next *add*.




# Algorithm model

## Example: simplified smartphone



# Functional specification languages

General purpose langages: ASSEMBLY, FORTRAN, PASCAL, C, etc.

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

These languages are more or less adapted to functional specification of algorithm that can describe potential parallelism and manage time.

# Functional specification languages

Synchronous languages: Esterel, Lustre, Scade, Signal, StateCharts, SyncCharts

**Synchronous languages** descend from CSP, CCS, TLA, etc., languages, allow the functional specification of potential parallelism (concurrence) and the management of time.

They have the following features:

- ESTEREL, STATECHARTS, SYNCCHARTS: Imperative, control flow, functional clock calculus, given maximal clock (*Tick*),
- LUSTRE, SCADE : Declarative, data flow, functional clock calculus, given maximal clock (*Tick*),
- SIGNAL: declarative, data flow, relational clock calculus, synthetized maximal clock.

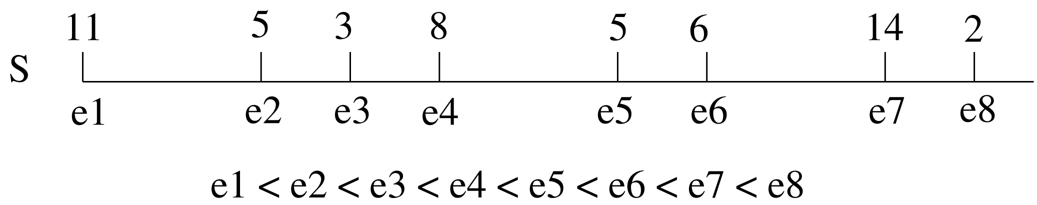
# Synchronous language SIGNAL

## Relations 1/4

**SIGNAL** is a **synchronous data flow** language for specifying relations between **valued events**, each event belongs to an infinite set of events called a **signal** and takes its values in a set such that real, integer, boolean, etc.

A signal is associated to each input (resp. output) of the control system, corresponding to the output (resp. intput) of a sensor (resp. actuator). A signal is also associated to each input and output of the other operations that specify the control system.
There are **four types** of relation:

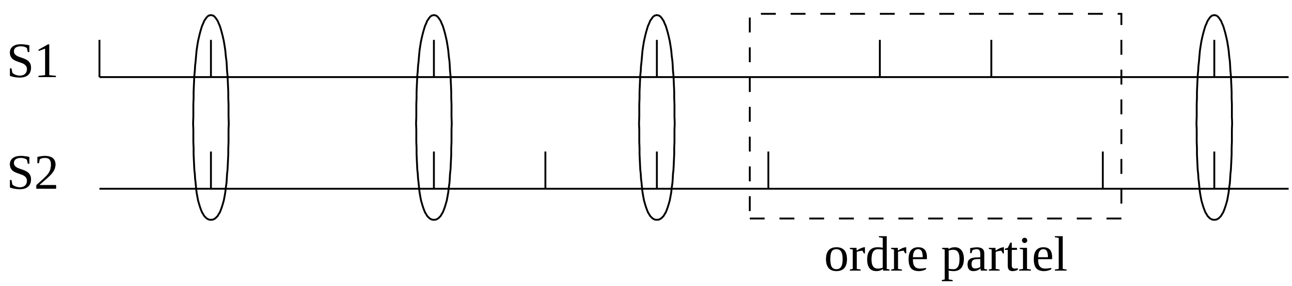
1. a **precedence relation** between two events of the same signal to associate a **logical instant** to each value. It is a strict total order relation that can be represented by a **logical timing diagram**:




# Synchronous language SIGNAL

## Relations 2/4

2. a **synchronism relation** between two events of two different signals. When two events are **synchronous** they are called **present** at the **same logical instant**. If one of the event is present while no other synchronous event on the other signal, that other event is said **absent**. This is an equivalence relation (reflexive, symmetric, transitive).

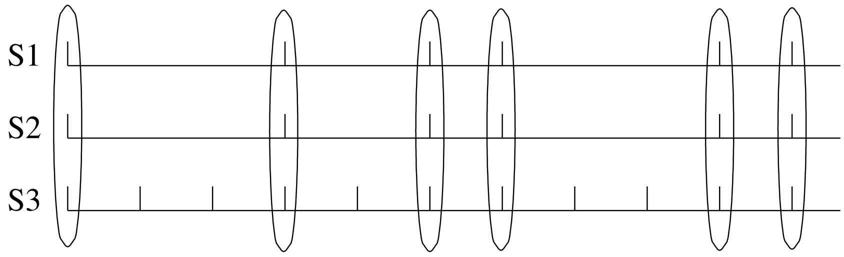



The synchronism equivalence relation between two events of two different signals combined with the strict total order relation between two events of the same signal, does not lead to a strict total order on the events of two different signal, but only to a **partial order**.

# Synchronous language SIGNAL

## Relations 3/4

3. a **synchronism relation** between two signals when every two events of these signals are synchronous, i.e. are present at the same logical instants (extension previous relation). It is also an equivalence relation. The set of the **synchronous** signal defines an equivalence class on the signal set, called their **clock**. These signals have the same clock.
   It is possible to define a **total order relation** $\geq$ on the clocks. The maximal clock defines the **logical time** of the system.

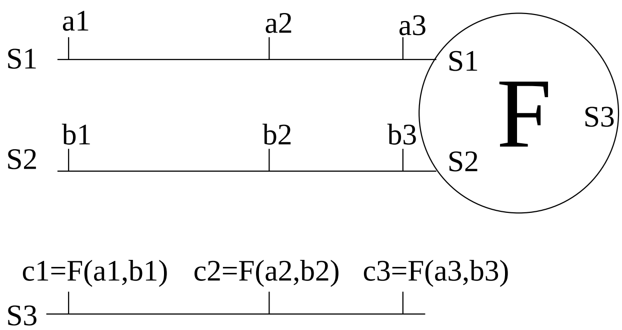



$S1$ and $S2$ have the same clock because they are synchronous. Some events of $S1$ and $S2$ are absent relatively to $S3$ whose clock is different from the clock of $S3$ which, such, is the greatest.

# Synchronous language SIGNAL

## Relations 4/4

4. **input-output relations** defining **4 types** of instruction.
   For example the **immediate function** between input and output signals of an operation, extension to signal of a classical function.




**Synchronous language hypothesis**: the output signals of an **immediate function** are synchronous with the input signals, that must also be sychronous. Such function is **causale** for each logical instant. A function **is not causale** if one of its output depends on itself (cycle). **The hardware architecture is not considered** during the functional specification. The notion of duration exists only by counting the events of a signal.

# Synchronous language SIGNAL

## Signal clock

The clock of a signal $X$ is denoted by $P(X)$. It is an ordered set of boolean values that are `True` when $X$ is **present** and `False` when $X$ is **absent**, **relatively to other signals**.

Two clocks can be compared with the **total order relation** $\geq$.

Thus, a boolean signal $B$ with values `True` or `False` has a clock $P(B)$ which can also take values `True` or `False`. $T(B)$ denotes the clock of the boolean signal $B$ when $B$ is `True`. For example we have
$P(X) \geq T(X < 0)$.

Since a clock is a boolean set, by denoting $\cap$ by " " and $\cup$ by "+" one can define the following basic relations on the clocks:
$P(X) = P(Y)P(Z) \qquad P(X) = P(Y) + P(Z) \qquad P(X) \geq P(Y)$

A SIGNAL program or **process** corresponds to the composition with the character **"|"** of instructions or **elementary processes** and/or processes (encapsulation, modularity). The **name identities** between output names and input names of instructions, induce precedences which define a partial order on the instruction execution.

# Synchronous language SIGNAL

## Four elementary processes

| Syntax | Clock equation | I/O Relation Types |
|---|---|---|
| **Immediate function**<br>(y1,..,yn) := f(x1,..,xm) | P(y1) =... P(yn)<br>= P(x1) =... P(xm) | y = f(x)<br>indifferent types |
| **Delay**<br>zx := x \$ n    zx init k | P(zx) = P(x) | zx(t) = x(t-1) pour n=1<br>x zx mêmes types |
| **Under-sampling**<br>y := x when b | P(y) = P(x) T(b) | y = x if b present true<br>x indifferent type boolean b |
| **Priority merge**<br>y := x0 default x1 | P(y) = P(x0) + P(x1) | y = x0 if x1 absent<br>y = x1 if x0 absent<br>y = x0 si x0 et x1 present<br>x0 x1 same types |

```
(| y1 := e when e<=0  %P(y1)=P(e)T(e<=0)=T(e<=0) as P(e)>=T(e<=0)%

 | y2 := e when e>0   %P(y2)=P(e)T(e>0)=T(e>0) as P(e)>=T(e>0)%

 | y3 := y1 + y2      %compilation error as T(e<=0)#T(e>0)%
 |)
```

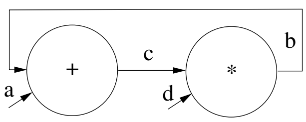
# Synchronous language SIGNAL

## Data flow graph of a process

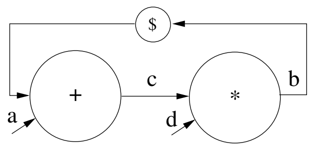
A SIGNAL process can be can represented by a **factorized conditioned repeated data flow graph** such that:

- every **vertex** with input and output **ports**, represents an elementaty SIGNAL process, associated to a **clock equation** defining the clocks of the output signals according to the clocks of the input signals,
- every **edge** or **data dependence**, between an input port and an output port, represents a signal auquel, associated to a clock,
- an edge sequence whose first vertex and the last vertex are identical lead to a cycle in the graph and thus to a **non causal** program. Cycles are **forbiden**. It is necessary to introduce at least a **delay** every the cycle ;

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




$\Rightarrow$




- an elementary process is executed according to the presence and absence of its input clocks defined by its clock equations.

# Synchronous language SIGNAL

## Compiler

The compiler performs some **verifications**:

- usual on values (type, table indices, division by zero, etc.),
- **formal**, guaranteeing **temporal logic properties**:
  - it verifies that every cycle contains a delay,
  - it verifies wether the clock equation system is correct, i.e. it tries to determine the clocks of the output signals according to the clocks of the input signal.
    If it is not possible to solve the clock equation system, there is two cases:
    - too much constraints (the clock calculus is redundant),
    - not enough constraints: a clock must be given to signals with undetermined clocks using a specific clock constraint instruction: `^=`, `P(X) = P(Y)` is associated to `X ^= Y`.

Finally, it generates a sequential program, for example in C, that allows a **functional and temporal logic simulation** guaranteeing the **correct order of events** for each output signal according to each input signal.

# Synchronous language SIGNAL

## Logical timing diagrams of signals 1/3

| Process | ··· | SIGNAL program control system: $\perp$= absent |
|---|---|---|
| X | > | 1.0 |
| Y | > | $\perp$ |
| Z | > | 2.5 |

**Elementary processes that do not modify clocks**

`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 |

# Synchronous language SIGNAL

## Logical timing diagrams of signals 2/3

`ZX := X $ 1 (ZX delayed of 1 logical instant w.r.t. X)`

| | | | | | | |
|---|---|---|---|---|---|---|
| X : | 2 | 1 | 5 | 4 | 3 | |
| ZX : | 0 | 2 | 1 | 5 | 4 | ZX initialized to 0 |

`^X (Clock of X: type event signal true=present false=absent)`

The type *event* is useful when one considers **only the clock** of a signal without considering the values of its events.

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

```
X ^= Y  => P(X)=P(Y) (clock constraint) X, Y type event
The indermined clock X takes the determined clock of Y
```

# Synchronous language SIGNAL

## Logical timing diagrams of signals 3/3

**Elementary processes that modify clocks: $\perp$= absent**

`X := A when B (subsampling of A by B true)`

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 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 (merge of X0 and X1, priority X0)`

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 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 |

# Synchronous language SIGNAL

## Constant signals

**The clock of a constant signal is determined by its context.**

No problem with immediat function and `when`:

```
X := A + 1 => P(X)=P(A)=P(1)
X := 1 when B => P(X)=P(1)T(B) : X value 1 clock B true
```

Because the right signal of a `default` has the priority **WARNING**:

```
Y := X0 default 1 => P(Y)=P(X0)=P(1) : Y=X0 clock of X0
Y := 1 default X1 => P(Y)=P(X1)=P(1) : Y=1 clock of X1
```

# Synchronous language SIGNAL

## Simplification rules used by the compiler

The clock set of a SIGNAL program with the partial order relation $\geq$, the binary relation **union**, additively noted defining the upper bound of two clocks and the binary relation **intersection**, multiplically noted defining the lower bound of two clocks, define a lattice with the following properties:

```
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)
```

One can deduce the following properties:

```
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) and, for example P(X) >= T(X=0)
P(X)P(Y)=P(X) if P(Y) >= P(X);  P(X)+P(Y)=P(X) if P(X) >= P(Y)
```

# Synchronous language SIGNAL

## Syntax of processes or programs 1/3

Underlined elements of the language (keywords) are terminal, bracketed elements "[ . . . ]" are optional, "/" means an alternative.

process $=$ <u>process</u> name <u>$=$</u> [ <u>{</u> parameters <u>}</u> ]<br>
        <u>(?</u> input-signals <u>!</u> output-signals <u>)</u><br>
        body<br>
        [ <u>where</u> local-signals<br>
            [ process <u>;</u> process . . . ]<br>
        ] <u>end</u><br>
process $=$ <u>function</u> name <u>$=$</u> <u>(?</u> input-signals <u>!</u> output-signals <u>)</u>

# Synchronous language SIGNAL

## Syntax of processes or programs 2/3

parameters = type name , name . . . ; type name , name . . .
input-signals = output-signals = type name , . . . ; type name , . . .
local-signals = type name [init val / expr-tableau] , name . . . ; . . .
type = [ [ val , . . . ] ] scalar-type
scalar-type = event / boolean / integer / real / dreal
body = (| inst | inst | . . . |)
inst = name := expr
inst = ( name , . . . ) := subprocess-call
inst = name := array-expr
inst = process-array
expr = **expression with elementary process and subprocess calls**
subprocess-call = name [ { val , . . . } ] ( expr , . . . )
nom = **sequence of alpha-numerical characters**
val = **expression containing only constants and/or parameters**

# Synchronous language SIGNAL

## Syntax of processes or programs 3/3

**% comments between percent %**

In an expression several elementary processes can be directly combined taking into account that an immediate function has a greater priority than a `when` which has a greater priority than a `default`.
Example :

```
y := a when b > 0 default z + 1 <=>
y := (a when (b > 0) ) default (z + 1)
```

The composition instruction "|" induces only a partial order on the process executions. Data dependences are deduced from the signal names. An ouptut signal is connected to an input signal with the same name, or to several input signals with the same names (data diffusion). Several output signals **cannot** be connected to a same input signal (**data confusion forbidden**).

# Synchronous language SIGNAL

## Syntax example

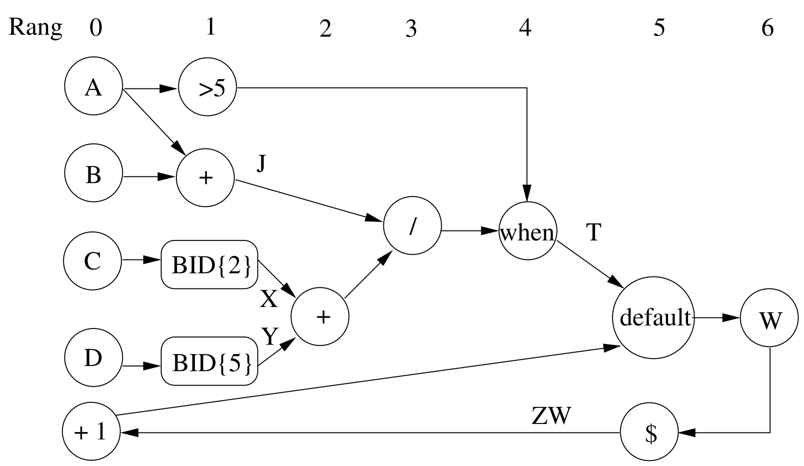
```
process EXAMPLE = {}
( ? integer A,B,C,D ! real W )
(| X := BID{2}(C)          %process BID parameter 2 P(X)=P(C)%
 | J := A+B                %P(J)=P(A)=P(B)%
 | T := (X+Y)/J when A>5   %P(T)=P(A)T(A>5)=T(A>5)%
 | Y := BID{5}(D)          %process BID parameter 5 P(Y)=P(D)%
                           %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
```

A parameter is an input **constant** signal that is internal/external to the

# Synchronous language SIGNAL

## Data flow graph of the process EXAMPLE

Signal program ⇔ Data flow graph.




Vertices without predecessors A, B, C, D (resp. without successors W) represent sensors (resp. des actuators) connected to other vertices by signals with same names as these sensors and actuators. **Clock constraints are not considered**.

Potential parallelism: $\{A, B, C, D, +1\}$ and $\{>5, +, BID2, BID5\}$
5 processors can be potentially in parallel.

# Synchronous language SIGNAL

## Process example 1/4

```
% Infinite counter on 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
```

# Synchronous language SIGNAL

## Process example 2/4

```
% Memorization of signal e with another signal m %
% output s takes values of input e when e %
% is present and takes the previous value of s %
% when e is absent %
% the clock of s must be greater than the clock of %
% the memorizing signal 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
```

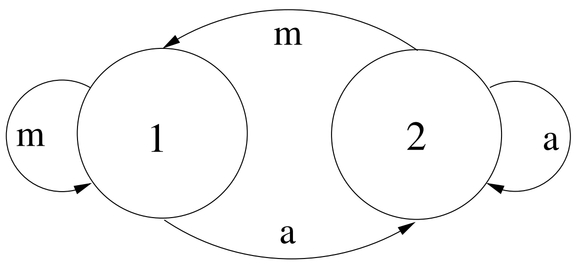
# Synchronous language SIGNAL

## Process example 3/4

```
% Counter on signal top, reset to zero on raz true %

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)
 |)                                      verifies (1) %
where integer zn init 0
end
```

# Synchronous language SIGNAL

## Process example 4/4




```
% Automaton 2 states start (1) and stop (2), two inputs %

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

# Multicomponent architecture specification

## General issues

### Distributed, parallel, multiprocessor, multicore architectures

These architectures are composed of several processors, connected by **point-to-point** or **multi-point** (bus) media, that communicate with **distributed memories** through **message passing** or with **shared memories**. Four possible pairs:

- **distributed**: communications are performed through message passing, processors are of different types (GRID),
- **parallel**: communications are performed with shared memories, processors are of same types,
- **multiprocessor**: communications are performed with shared memories, processors are of same or different types,
- **multicore**: processors are located on the same chip and communicate with shared memories and/or network on chip.

The way processors and media are connected leads to different **topologies**: ring, star, mesh, hypercube, totally connected, etc. A **route** is a chain starting and ending with a processor, including alternatively a processor then a communication medium.

# General issues

## Parallelism, multicomponent architecture

**Parallelism types** (Flynn's classification 1996):

- **control**: (MIMD) processors execute different on different data, it is limited by dependences,
- **data**: (SIMD, SPMD) processors execute the same computation on different data,
- **flow**: (MISD) pipe-line, processeurs execute different computation on the same data.

**Multicomponent architecture:** heterogeneous, including processors of different types and specific integrated circuits of different types. They are connected by communication media of different types, offering different types of parallelism.

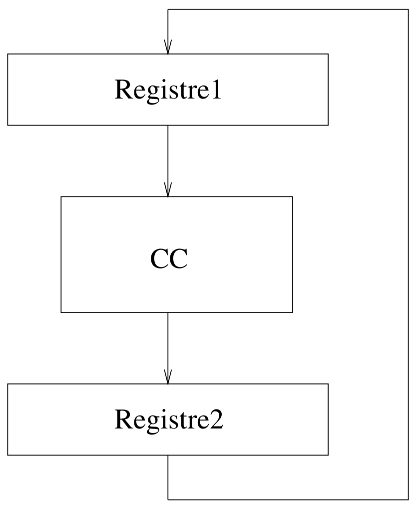
- **Processor**: programmable component which executes sequentially instructions of a program.
- **Specific integrated circuit** (ASIC, FPGA): non programmable component which executes **only one** operation.
- **Communication medium**: point-to-point or multi-point (bus) connections.

# Multicomponent architecture model

## RTL

**RTL model: Register Transfert Level**

Data transfers between registers through a combinatorial circuit (CC).




- Computation - CC
- Memory (cache, extern) - Register
- Communication medium - Data path - $\longrightarrow$

# Multicomponent architecture model

## Sequential machine

A Multicomponent architecture is based on the notion of **sequential machine**.

- **Finite automaton** (acceptor or recognizer): finite state machine (FSM) $(E, X, i, f, t)$

  $E$ finite set of input symbols (input events)
  $X$ finite set of states, initial states $i \subset X$, final states $f \subset X$
  $X$ is associated to a memory containing the past of the automaton
  $t$ transition function $t : E \times X \rightarrow X \qquad x_{t+1} = t(e, x_t)$

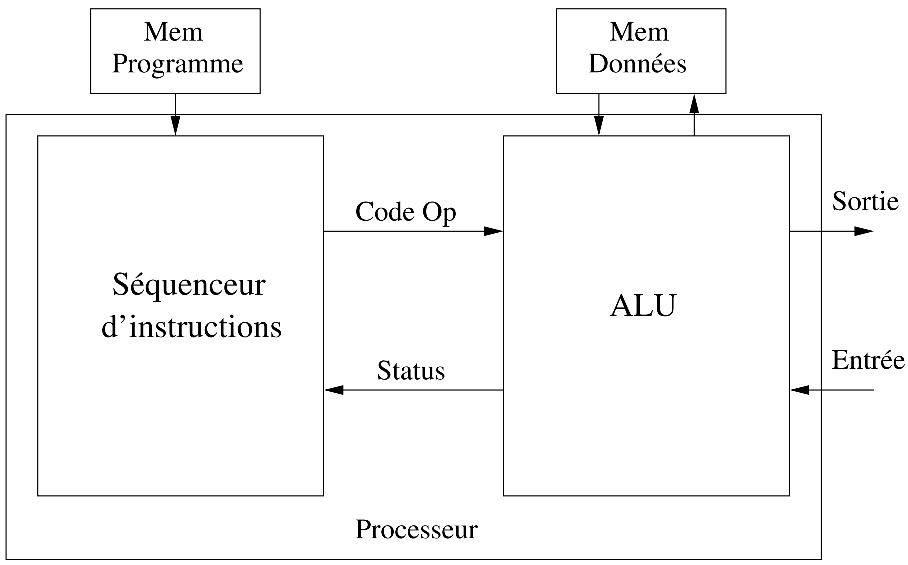
- **Finite automaton with outputs** (transductor): sequential machine in digital electronic device domain $(E, X, i, f, t, S, s)$

  $S$ finite set of output symbols (output events)
  $t$ transition function, $s$ output function which executes the CC
  Mealy $s : E \times X \rightarrow S \quad w = s(e, x_t) \quad$ Moore $s : X \rightarrow S \quad w = s(x_t)$

# Multicomponent architecture model

## Processor

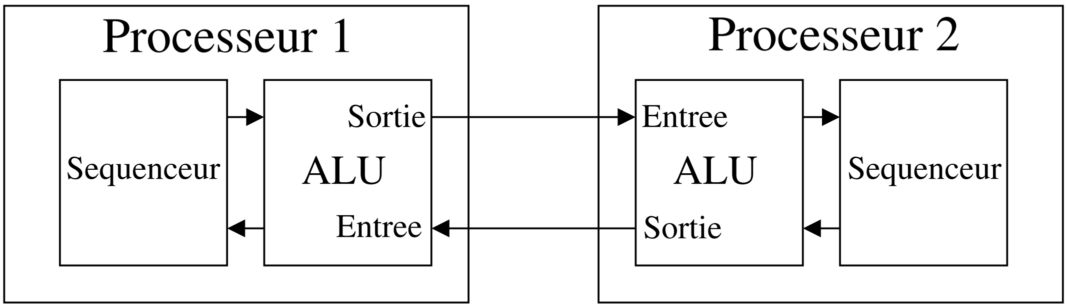
Processor = two sequential machines connected: sequencer and ALU.
**The sequencer** reads the state produced by the ALU, reads an instruction in the program memory and writes an operation code in the ALU.
**The ALU** reads an operation code and executes the corresponding operation which reads and writes data in the data memory.



# Multicomponent architecture model

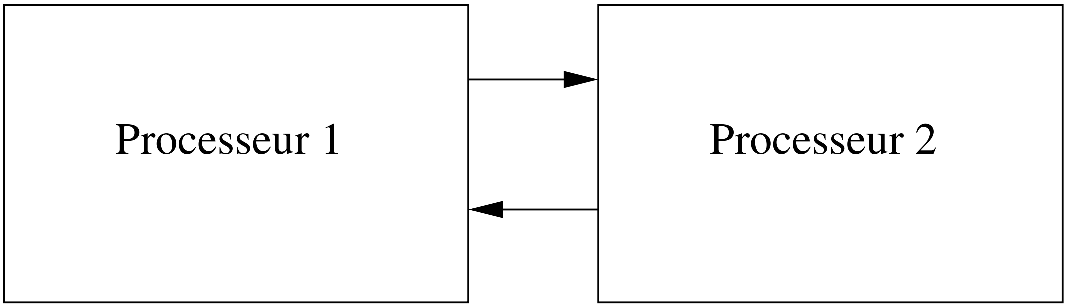
## Simple model of distributed or parallel architecture

Two processors, composed each of them of two sequential machines, connected by point-to-point links, constitute a parallel architecture.




**Abstraction**

$\Downarrow$




# Multicomponent architecture model

## AAA multicomponent model 1/2

- **vertex**: atomic sequential machine of **four types** :
  1. **operator** sequences operations (ALU, FPU . . . ) and data dependences when there is no communicator,
  2. **communicator** sequences data dependences (DMA . . . ),
  3. **memory** :
     - random access (RAM):
       - data (D) or program (P),
       - for data communications shared by the communicators,
     - sequential (SAM): for data communications only, distributed on the communicators, by message passing point-to-point or multi-point (bus), with or without diffusion.
  4. **mux, demux**:
     - **mux**: access of several operators and/or communicators to a shared memory => arbitration,
     - **demux**: access of one operator and/or one communicator to several memories => routing.
- **edge**: bidirectional connection between two vertices, composed of two opposite direction directed edges, such connection cannot connect two vertices of the same type except for mux/demux vertex type.

# Multicomponent architecture model

## AAA multicomponent model 1/2 and abstraction 1/2

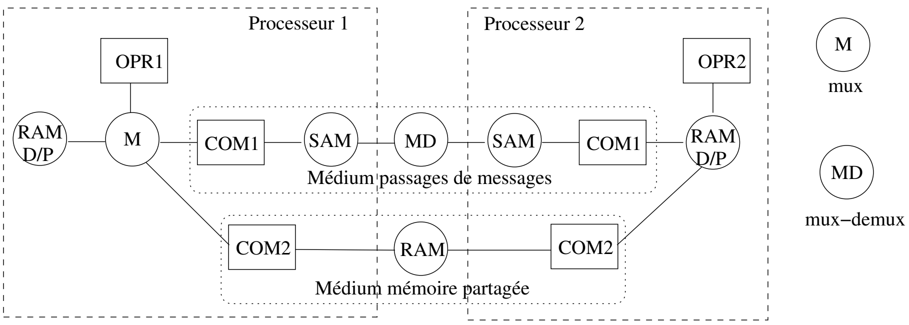
**Processor**: graph with of only one operator and several mux/demux, **RAM P**, RAM D, communicators, SAM.
**Specific integrated circuit**: graph with of only one operator and several mux/demux, RAM D, communicators, SAM.
**Message passing medium**: graph with communicator, SAM, mux, demux
**Shared memory medium**: graph with communicators, RAM.
**Bus**: connection of several processors.
**Router**: connection of a processor and several routers (N,S,E,O).

**Abstraction**

A processor or an integrated circuit subgraph can be **abstracted** in a unique operator vertex. Data and program memories are hiden. A medium subgraph can be **abstracted** in a unique operator medium. For the message passing media each communicator with its distributed SAM is represented by a port of the operator, the mux/demux is kept. For the shared memory media only each communicator is represented by a port of the operator.

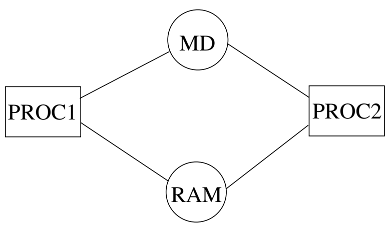
# Multicomponent architecture model examples

## Abstraction 2/2




**Abstraction**

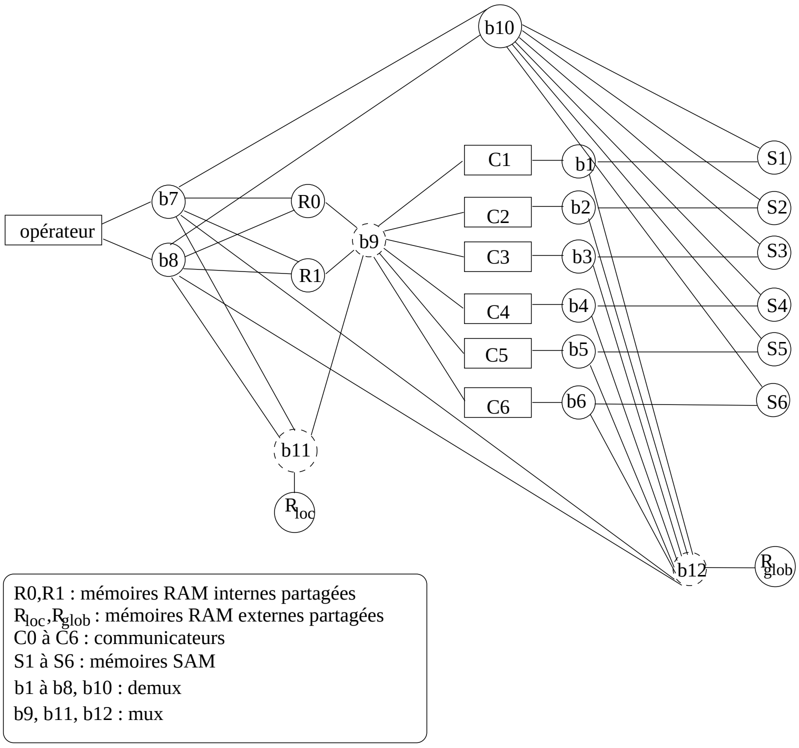
⇓




Abstraction leads to a smaller graph thus to a faster but less accurate optimizations.

# Multicomponent architecture model examples

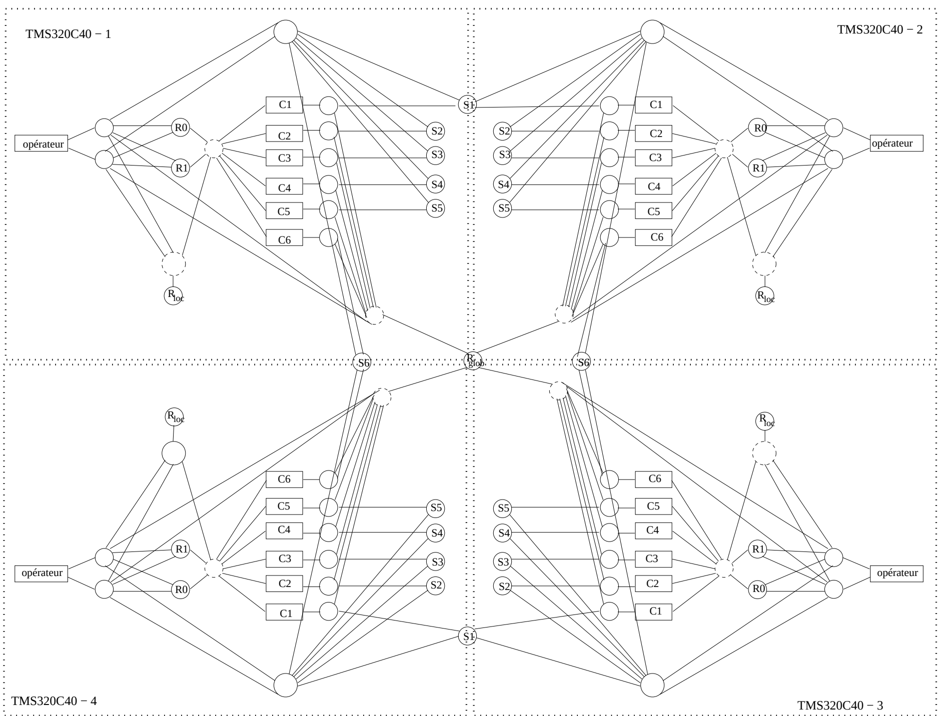
## TMS320C40




29 vertices in the architecture graph

# Multicomponent architecture model examples

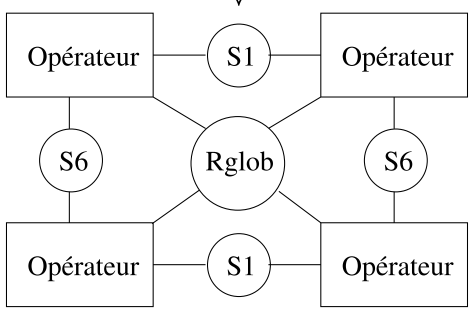
## Four TMS320C40 connected by point-to-point and multi-point media




109 vertices

**Abstraction**

⇓

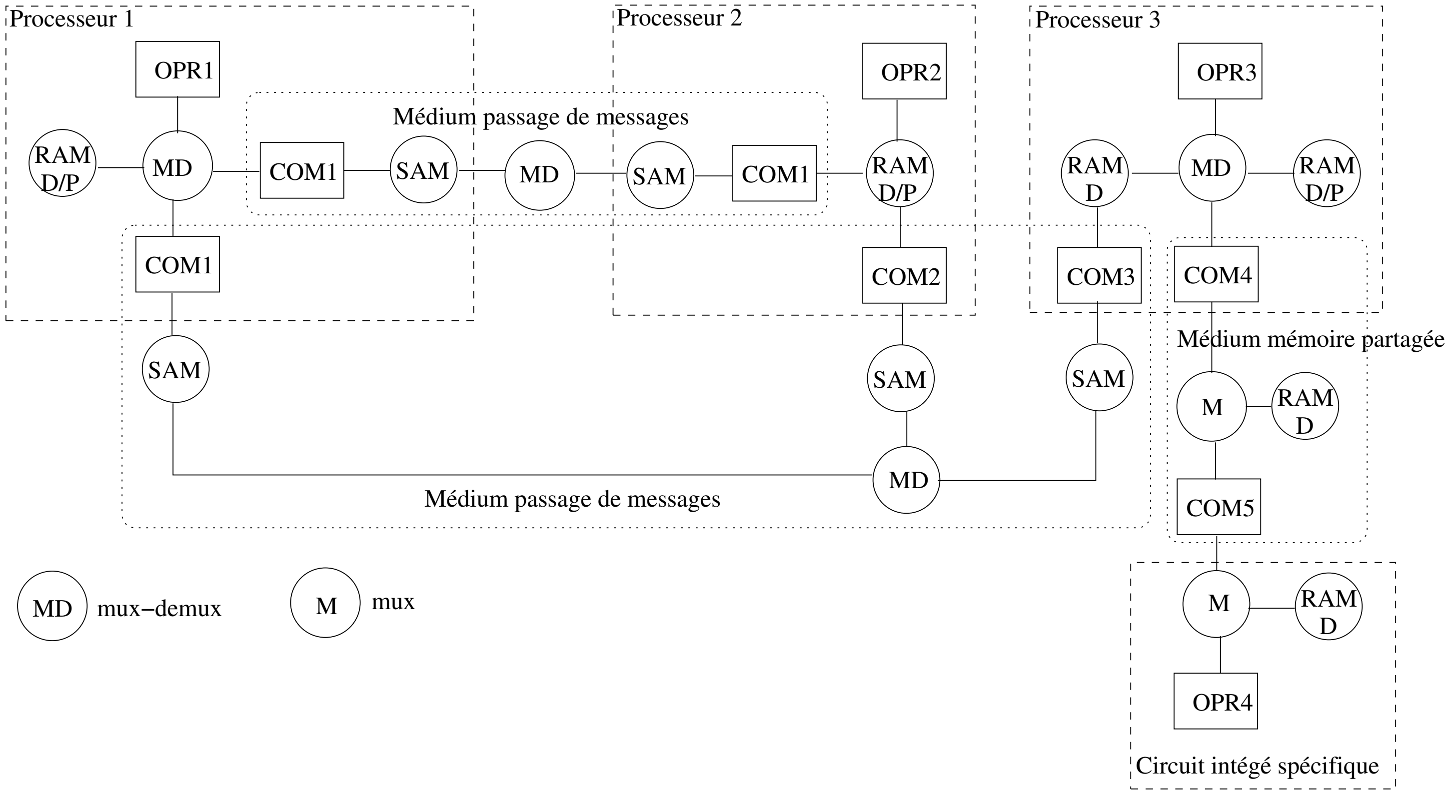



(D/P) and mux/demux are hiden, communicators are operator ports.

9 vertices

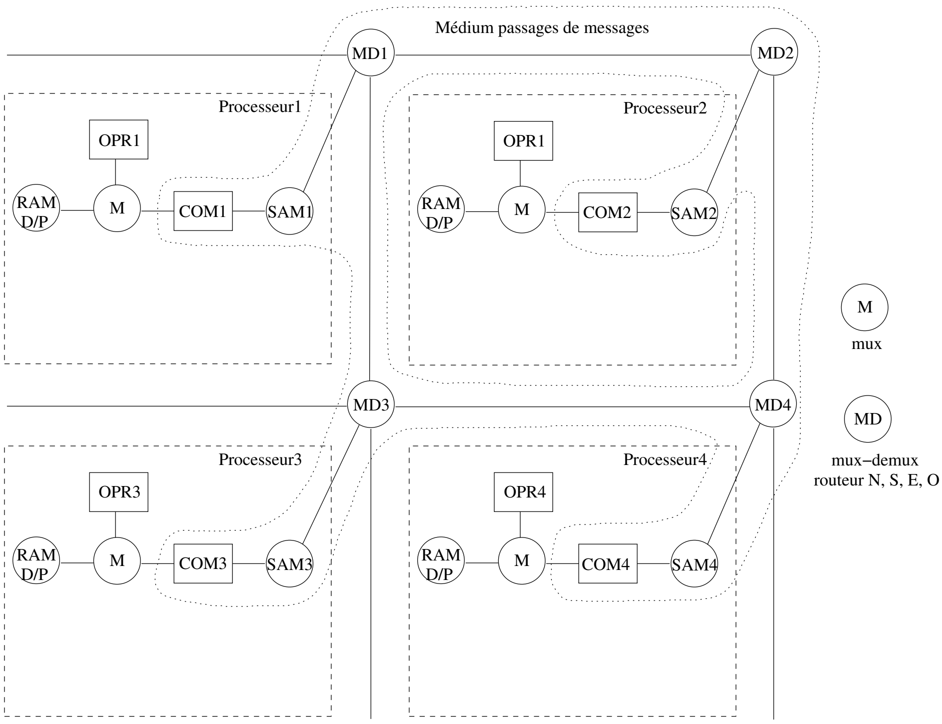
# Multicomponent architecture model examples

## Three processors and a specific integrated circuit connected by point-to-point and multi-point media




# Multicomponent architecture model examples

## Four processors connected by a network




Message passing medium $\{(com1, sam1), (sam1, md1), (com2, sam2), (sam2, md2), (com3, sam3), (sam3, md3), (com4, sam4), (sam4, md4), (md1, md2), (md1, md3), (md2, md4), (md3, md4)\}$

# Optimized implementation

## General issues

### Distribution and scheduling 1/2

The **implementation** is achieved from the functional and non-functional (multicomponent architecture, timing characteristics dependent or independent from the architecture) specifications. It consists of a distribution and a scheduling of the algorithm on the architecture.

The distribution allocates operations to operators (computation resources) and data dependences to media (communication resources). The distribution is also called: allocation, partitioning or placement.

For each operator, the **scheduling** consists in determining in which order the operations will be executed on this operator, and for each medium in which order the data dependences will be executed on this medium.

The scheduling must **preserve** the **partial order related to dependences** and must **guarantee** **real-time constraints** are met.

# General issues

## Distribution and scheduling 2/2

Distribution and scheduling can be achieved **online**, during the execution of the application, or **offline** before the execution of the application.

Online approaches can take into account operations whose timing characteristics are not totally known during the specification, however they have an important overhead, and are not deterministic.

Offline approaches require an accurate knowledge of the algorithm operations and of the multicomponent architecture, however they have a small overhead and are deterministic, well suited to critical hard real-time.

Later on we shall favour the **offline approaches** whose results can be used to generate automatically **dedicated executives** that, possibly, may call a **resident executive**.

# Uniprocessor real-time scheduling

## Classical approach 1/9

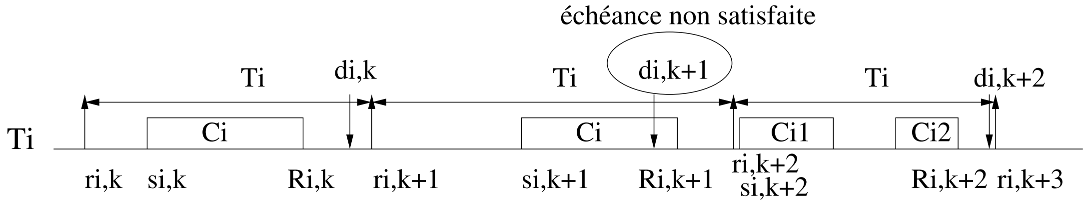
The classical preemptive task model, proposed by Liu and Layland in 1973, is based on the utilization of an inline executive for which each task $i$ is the repetition of an "instance" or "job" indexed by $k = 1..\infty$.

A **task** is an **operation** with the following characteristics:

- **release time** $r_i^k$, time when the task is activated, possibly periodic (infinite repetition) with period $T_i$, thus $r_i^k = r_i^0 + kT_i$,
- **first release time or offset** $r_i^0$,
- **start time of execution** $s_i^k \neq r_i^k$,
- **worst case execution time WCET** $C_i$, to which is added an **approximation of the executive cost**,
- **relative deadline or critical delay** $D_i$, duration from $r_i^k$, before the task $i$ **must be completed**,
- **absolute deadline** from the time origin $d_i^k = r_i^k + D_i$,
- **response time** $R_i^k$, duration from $r_i^k$ where the task $i$ is **completed**,
- **laxity** $l_i^k(t) = d_i^k - (t + C_i(t))$, difference between the absolute deadline and the duration that is already executed.

# Uniprocessor real-time scheduling

## Classical approach 2/9



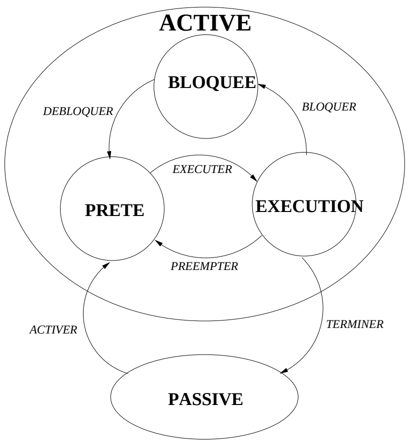
Modèle de tâche de Liu et Layland

A task is **schedulable** if all its instances satisfy their deadline.
A set of tasks is **schedulable** if every task is **schedulable**.
The **scheduler** of the executive executes a, possibly, preemptive **real-time scheduling algorithm** based on **priorities**.
The **preemption** allows a task to be interrupted by another task with a higher priority. It increases the number of possible schedulings, but it is necessary to account carrefully its **cost** inside the cost of the executive in order to garantee that the schedulability conditions are **satisfied**.
**Fixed priorities** are used by the scheduler to choose the next task to execute do not change during the execution.
**Dynamic priorities** may change when a task is activated or when it completes, leading to a higher scheduler cost.

# Uniprocessor real-time scheduling

## Classical approach 3/9

**The scheduler is composed of an automaton for each task and a specific automaton that manages task automata**




A task automaton has **four states**:

**PRETE** to be executed because it just has beeen activated (released),
**EXECUTION** executing,
**BLOQUEE** waiting for a resources,
**PASSIVE** waiting for an *activation*.

A task automaton has **six input events**:

*ACTIVER*: produced by the external interrupt associated to the task,
*TERMINER*: produced by the task itself when it completes,
*EXECUTER*, *PREEMPTER*: produced by the **manager automaton**,
*BLOQUER*, *DEBLOQUER*: produced by the **manager automaton**.

# Uniprocessor real-time scheduling

## Classical approach 4/9

**Only one** of the task automata is in the state **EXECUTION**

**Fixed priority scheduling**

When the event *ACTIVER* of a task occurs, this task goes from the state **PASSIVE** to **PRETE**, the **manager automaton** or an hardware device external to the processor **compares** it priority to the priority of the task being executed, the only task to be in the state **EXECUTION**.

If the priority of the activated task is greater to the one of the task being executed, the **manager automaton** saves its context, and then produces the event *PREEMPTER* for its automaton which goes from the state **EXECUTION** to **PRETE**, it produces an event *EXECUTER* for the automaton of the activated task which goes from the state **PRETE** to **EXECUTION** and finally executes the activated task.

# Uniprocessor real-time scheduling

## Classical approach 5/9

When the event *TERMINER*, produced by the task being executed, occurs then this task goes from the state **EXECUTION** to **PASSIVE**, the **manager automaton** **compares** the priorities of the tasks in the state **PRETE** and produces the event *EXECUTER* for the automaton of the task with the highest priority which goes from the state **PRETE** to **EXECUTION**. If this task has been preempted the **manager automaton** restores its context and resumes its execution, otherwise it starts a new execution of an instance.

The **manager automaton** produces the event *BLOQUER* when a task cannot access to a shared resource already used by another task. It produces the event *DEBLOQUER* when the blocked task can access the resource, and goes from the state **BLOQUEE** to **PRETE**.

# Uniprocessor real-time scheduling

## Classical approach 6/9

The **feasibility analysis** of a real-time task set consists in finding conditions such that all the tasks **satify** their constraints. The **schedulability analysis** consists in finding such conditions but with a given scheduling algorithm.

We consider a set of $n$ **preemptive periodic independent** tasks whose scheduler and preemption costs are approximated in the WCET, the utilization factor is $U = \sum_{i=1}^{n} C_i/T_i$ and the density is $\Delta = \sum_{i=1}^{n} C_i/D_i$.

Here are the **schedulability conditions** according to the better known **scheduling algorithms**:

- fixed priorities (do not change during the task execution):
  - **RM** (Rate Monotonic) algorithm: priority inversely proportional to the period, the tasks are schedulable **if** $U \leq n(2^{1/n} - 1)$, $D_i = T_i$,
  - **DM** (Deadline Monotonic) algorithm: priority inversely proportional to the deadline, the tasks are schedulable **if** $\Delta \leq n(2^{1/n} - 1)$, $D_i \leq T_i$,

# Uniprocessor real-time scheduling

## Classical approach 7/9

- dynamic priorities (determined at activation and completion times):
  - **EDF** (Earliest Deadline First) algorithm: priority to the task with the smallest absolute deadline, *dynamic between instances, fixed inside an instance*, the tasks are schdedulable **if and only if**: $U \leq 1$, $D_i = T_i$, the tasks are schdedulable **if**: $\Delta \leq 1$, $D_i \leq T_i$,
  - **LLF** (Least Laxity First) algorithm: priority to the task with the smallest laxity, *dynamic between instances, dynamic inside an instance*, the tasks are schdedulable **if** same conditions than EDF.

When considering **dependent tasks**, dependences are due to:
- **precedences only**, this case can be reduced to the non dependent tasks case by adding new constraints that modify release times and dealines,
- **data transfers**, in addition to the precedence constraint, it is necessary to manage data shared between the producer task and the consumer task, as well as the data exchanges according to the respective values of their periods.

## Uniprocessor real-time scheduling
### Classical approach 8/9

Preemptions may involve **priority inversions** when **several dependent tasks share a data**. The automaton of a task accessing a shared data, goes in the state **BLOQUEE** while this one is not available. If several tasks are in this state, some **deadlocks** may occur.
This problem can be solved with two protocols:

- **priority inheritance**: the task which is executing while accessing a data, inherits the highest priority of the tasks which share this data such that it releases the data as soon as possible, thus it is possible to compute its maximum blocking time,
- **priority ceiling**: in order to prevent deadlocks, the previous protocol is extended by adding a priority to each data equal to the highest priority (ceiling) of the tasks which share this data, such that a task cannot access a data only if the priority of this data is higher than the priorities of the other data that the other tasks may access.

## Uniprocessor real-time scheduling
### Classical approach 9/9

EDF and LLF algorithms can be used to schedule a set of **aperiodic tasks**.

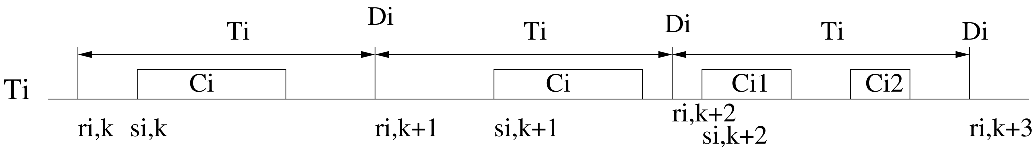
It is also possible to use the following algorithms:

- **background**: tasks are scheduled when the processor has no periodic tasks to execute,
- **task server**: tasks are scheduled by an additional periodic task which executes some parts of the aperiodic tasks,
- **slack stealing**: tasks are scheduled during laxities of the periodic tasks.

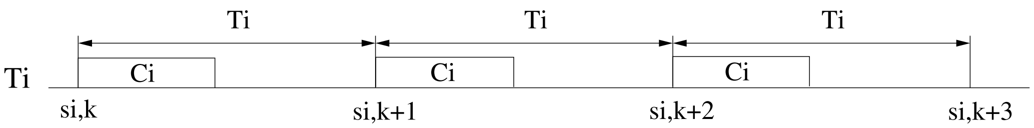
# Uniprocessor real-time scheduling

## AAA 1/3

Critical real-time applications are composed of tasks corresponding to sensors, actuators, and control processes which must not have **jitter**. It is the reason why they must have a **strict period** and must be non preemptive such that their **response time** do not vary being equal to their WCET. In order to simplify the problem we consider that all the tasks have these characteristics following the **"AAA"** model where a task $o_i$, called **operation**, is infinitely repeated with a **stric period** with a WCET $C_i$ including a fixed cost of the executive.




Modèle de tâche de Liu et Layland




Modèle d'opération AAA

# Uniprocessor real-time scheduling

## AAA 2/3

In this model an operation $o_i$ **has no** release time but only a **start time** $s_i^k = s_i^{k-1} + T_i = s_i^0 + k * T_i$ for every instance $k$. Its relative deadline **is equal to** its period imposing that $C_i \leq T_i$.

At every release and completion of a task, the executive cost is composed of:

- the scheduler cost:
  - offline: reading in a table the task to execute,
  - online: choice of task by comparing their priorities,
- cost of the preemptions **able to involve other preemptions**
  - offline: no preemption,
  - inline: store and restore of every task context.

We choose the **non preemptive** case even though schedulability analyses are more complicated and it reduces the scheduling possibilities, because the preemption cost is equal to zero. In addition we choose **offline executive** because the scheduler cost is smaller than the cost of inline executive.

## Uniprocessor real-time scheduling

AAA 3/3

Aperiodic operations are **made periodic** by pooling the events that trigger them at a period which is smaller than the minimum delay between two of their occurences.

We want to solve a **non preemptive** uniprocessor scheduling problem of operations that must satisfy constraints of dependence and deadline equal to its strict period.

**Sufficient feasibility condition** (Korst 1991 for two tasks, Kermia-Sorel 2009 for more than two tasks): a dependence graph of $n$ non preemptive operations $o_i$ with WCET $C_i$ and strict period $T_i$ is schedulable **if** $\sum_{i=1}^{n} C_i \leq GCD(T_i)$.

## Multiprocessor real-time scheduling

classical approach

There are two main approaches that minimize the utilization factor $U$:

- **global:** a unique scheduler for all the processors which can migrate tasks from one to another processor. The migration cost is very high with the current processors. This is a theoretical problem solved when preemption and migration costs are equal to zero with an optimal algorithm called "Pfair",
- **partitioned:** one scheduler for each processor where some tasks were distributed such that these tasks are schedulable. Minimizing the utilization factor is an **NP-hard** problem equivalent to a "Bin Packing" problem which consists in filling bins of same size with objects of different sizes. This problem can be solved in a reasonable time only with **heuristics** that procuce non optimal (approximated) solutions. "First Fit", "Next Fit", "Best Fit", "Worst Fit", heuristics distribute the tasks on the processors while verifying classical schedulability conditions (RM, DM, etc.). Generally, the scheduling of the interprocessor communications is achieved separately, often without accounting their cost.

# Multiprocessor real-time scheduling

## AAA

Due to the prohibitive cost of migration, we choose the **partitioned scheduling** approach. We have to solve a distribution problem for the different processors and for each processor a non preemptive scheduling problem of operations (tasks) that must verify **dependence constraints** and **deadline equal to a strict period constraints**. In addition, we want to minimize the **total execution time** (makespan) while considering the **interprocessor communication costs**.

A distribution and a scheduling is obtained by **transforming** the algorithm graph **according to** the architecture graph, assuming that all the possible routes are known. This amounts to **reduce the potential parallelism** of the algorithm (scheduling) such that it corresponds to the **actual parallelism** of the architecture (distribution).

**An optimized implementation** is obtained by seeking among all the possible transformations, one which **minimizes the total execution time** called **"application latency"** = Max(input-output latencies).

# Formalization of the AAA implementation

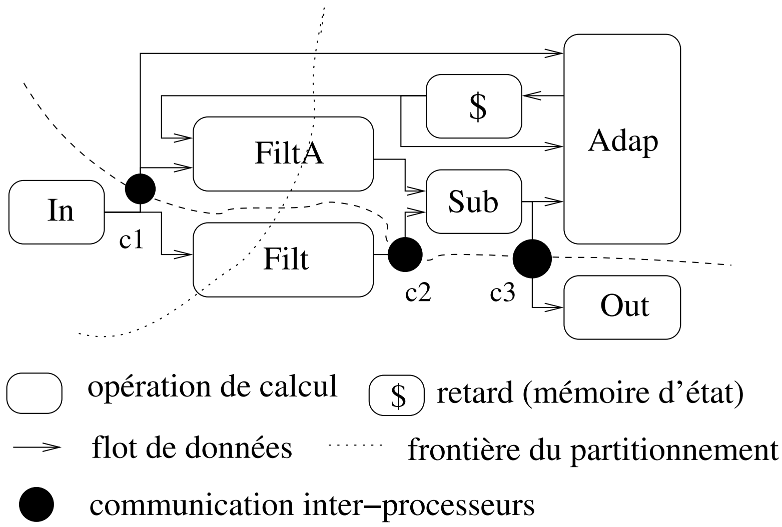
## Algorithm graph transformations according to the architecture graph

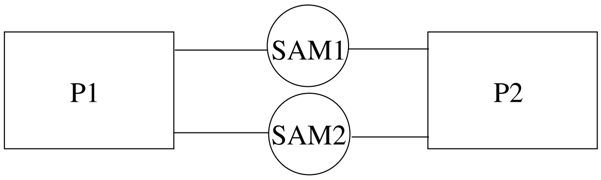
The algorithm graph is transformed as follows:

- **partition** the operation set in as many elements as there are operators in the architecture graph,
- **replace** edges relating different partition elements by as many **new communication vertices and edges** as there are media in the route on which these communication operations are distributed,
- **add** precedence edges between operations distributed on a processor but not yet related by data dependences,
- **add** precedence edges between communication operations distributed on a medium but not yet related by data precedences.

# Formalization of the AAA implementation

## AAA distribution and scheduling example




{In, Filt, Out} distributed on P1 and {FiltA, sub, Adap} distributed on P2.
{c1} distributed on SAM1 and {c2, c3} distributed on SAM2

# Formalization of the AAA implementation

## Principles

The AAA implementation (distribution and scheduling) is a graph transformation **formalized by the composition of three relations** each of them relating two pairs of graphs (algorithm, architecture):

- **routing**: complete connection of the architecture graph,
- **distribution** of the algorithm operations on the operators,
- **distribution** of the communication operations induced by the previous distribution on the media,
- **scheduling** of the operations on the operators where they were distributed, and of the communication operations due to data dependences relating operations distributed on different operators, on the media where they were distributed.

The number of distributions and the number of schedulings obtained from a given pair (algorithm, architecture) **may be very large but it is finite**. This relation composition **preserves temporal logic properties** guaranteed during formal verifications of the functional specification.

# Formalization of the AAA implementation

## Routing relation

### Routing

**Determination of all the paths**. A path (routes) in the architecture graph is a sequence of vertices related by edges, involving a total order.

$\mathcal{P}$ = set of processors, $Card(\mathcal{P}) = p$
$\mathcal{L}$ = set of communication media, $Card(\mathcal{L}) = l$
$\mathcal{X}$ = set of connections, $x = (p, l)$ ou $(l, p)$, $p \in \mathcal{P}$ et $l \in \mathcal{L}$,
$\mathcal{R}$ = set of paths of $(\mathcal{P} \cup \mathcal{L}, \mathcal{X})$, $r = (p, l, p', l', p''), p, p', p'' \in \mathcal{P}$ et $l, l' \in \mathcal{L}$

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

# Formalization of the AAA implementation

## Distribution relation 1/2

### Distribution

$\mathcal{O}$ = set of operations, $Card(\mathcal{O}) = n$
$\mathcal{D}$ = set of data dependences, $d = (o, o'), o, o' \in \mathcal{O}$

**Distribution of operations on operators** = partition of the $n$ operations of $\mathcal{O}$ in $p$ elements, $n > p$. The number of possible partitions is **computable** equal to: $\displaystyle\sum_{k=0}^{p} (-1)^k \frac{(p-k)^n}{(p-k)!k!}$
For example with $n = 4$, $p = 2$ we have 7 possible partitions, with $n = 12$, $p = 3$ we have 86 526 and with $n = 12$, $p = 5$ we have 1 379 400.

$\mathcal{O} \supset \mathcal{O}_p$ = set of operations executed on processor $p$
$\mathcal{D} \supset \mathcal{D}_p$ = set of data dependences between operations executed by $p$
$\mathcal{D} \supset \mathcal{D}_r$ = set of inter-partition data dependence

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

# Formalization of the AAA implementation

## Distribution relation 2/2

**Distribution of communication operations on media** = partition of $\mathcal{D}_r$ in Card($\mathcal{L}$) elements whose number is **computable**.

Every inter-partition dependence $(o_{i\,p_i}, o_{j\,p_j})$, with $o_i$ on $p_i$, $o_j$ on $p_j$, is transformed in a path (total order) including a vertex for each medium $m$ of the route on which it was distributed:
$\forall r \in \mathcal{R}, \forall d_r \in \mathcal{D}_r \quad d_r \xrightarrow{com} \left(o_{i_{p_i}}, o_{l_1}, o_{l_2}, \ldots o_{l_{k-1}}, o_{l_k}, \ldots o_{l_m}, o_{j_{p_j}}\right)$

A vertex $o_l$ is a new **communication operation** distributed on the medium $l$. An edge $(o_{l_{k-1}}, o_{l_k}) = c_p$ is a data dependence distributed on the processor $p$.
We group the $o_l$ of a same $l \in \mathcal{L}$ in the set $\mathcal{O}_l$ and the $c_p$ of a same $p \in \mathcal{P}$ in the set $\mathcal{C}_p$ with $\mathcal{C}_p = \mathcal{C}_{p(calc,com)} \cup \mathcal{C}_{p(com,com)} \cup \mathcal{C}_{p(com,calc)}$

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

# Formalization of the AAA implementation

## Scheduling relation

The addition of communication operations $o_l$ and their distribution on the media **do not modify the partial order** $D$ of the algorithm graph. The number of vertices and edges increases according to the number of media.

**Scheduling**

On each processor $p$, a scheduling of the computation operations is a total order $\bar{\mathcal{D}}_p$ which includes the partial order $\mathcal{D}_p$, $\mathcal{D}_p \subseteq \bar{\mathcal{D}}_p$
Similarly on each medium, a scheduling of the communication operations is a total order $\bar{\mathcal{D}}_l$ which includes the partial order $\mathcal{D}_l, \mathcal{D}_l \subseteq \bar{\mathcal{D}}_l$

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

The number of edges increases according to the number of non dependent operations. The number of total orders, obtained from a partial order, is **computable** and equal to $C_n^2 = \dfrac{n!}{2!(n-2)!}$ for $n$ non dependent operations. We have $C_{10}^2 = 45$.

# Formalization of the AAA implementation

## Composition of three relations

**Implementation = Routing** $o$ **Distribution** $o$ **Scheduling**

The implementation (distribution and scheduling) is the graph transformation *dist/sched* formalized by the composition of the relations:

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

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

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

Since the number of partitions and the number of total orders obtained from a partial order is computable, thus **the number of possible implementations is computable**.

# Formalization of the AAA implementation

## External composition law

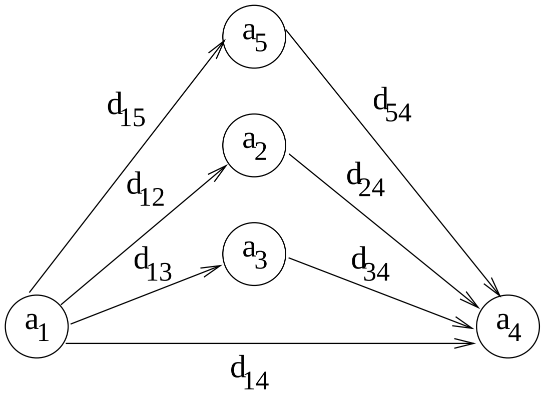
When assuming that all the possible routes of an architecture is known, this composition of three relations may be seen as an external composition law denoted $*$. Let $G_{al}$ be the algorithm graph set and let $G_{ar}$ be the architecture graph set, thus we have:

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

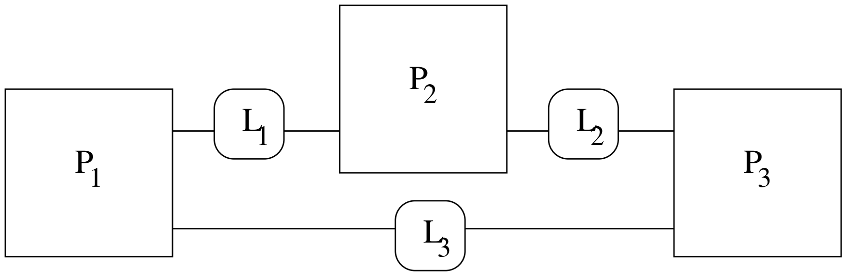
We choose among all the possible graph transformations, the ones which correspond to **valid implementations**, that is, for which the resulting partial order is **compatible** with the initial partial order of the algorithm graph, and which **do not introduce cycles** in a path of computation vertices that do not contain a delay vertex. This situation could lead to a deadlock. On the other hand, cycles on communication operations are allowed if they do not introduce cycles on computation operations.
"**compatible** with the initial partial order of the algorithm graph" means that no vertex and no edge were suppressed, that only edges were added, and finally that vertices and edges were added as paths (total order), only to replace inter-partition edges.

# Formalization of the AAA implementation

## Example 1/3



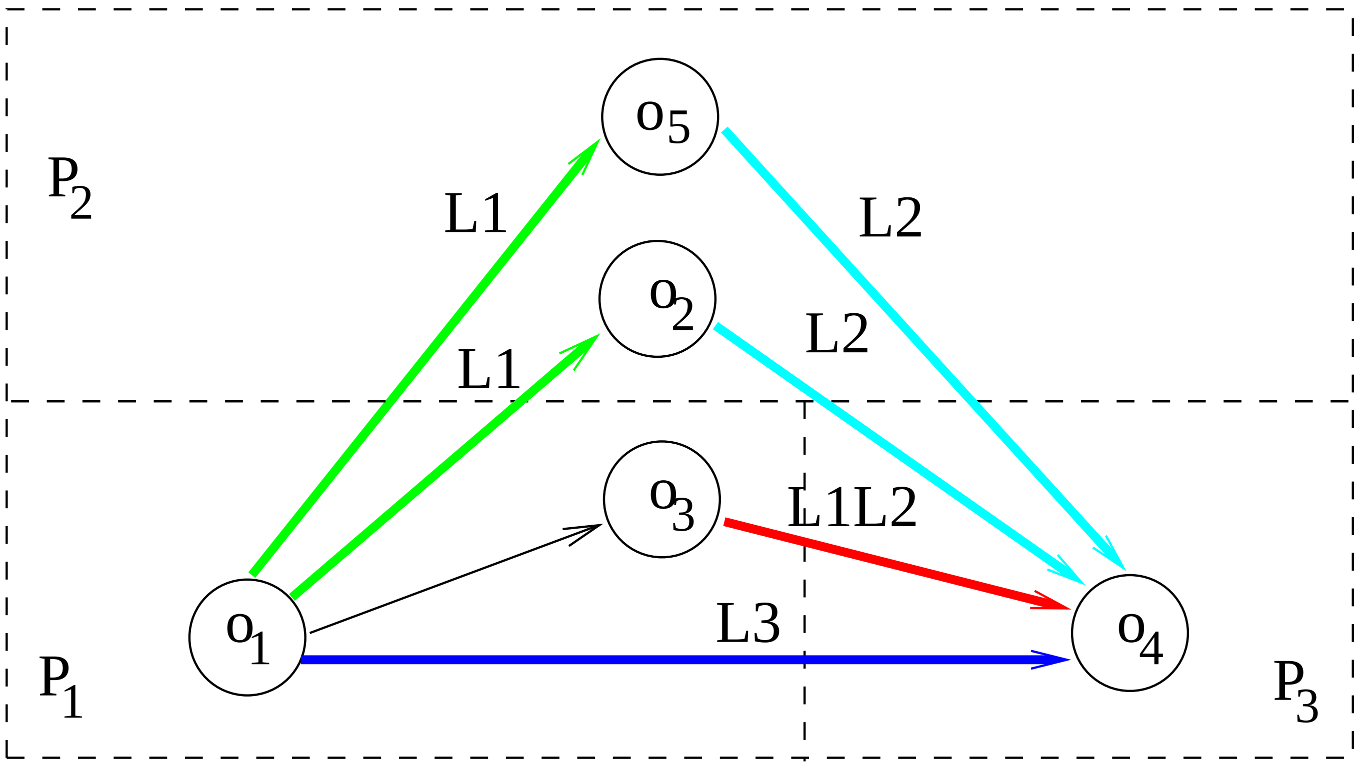

Algorithm graph




Architecture graph

# Formalization of the AAA implementation

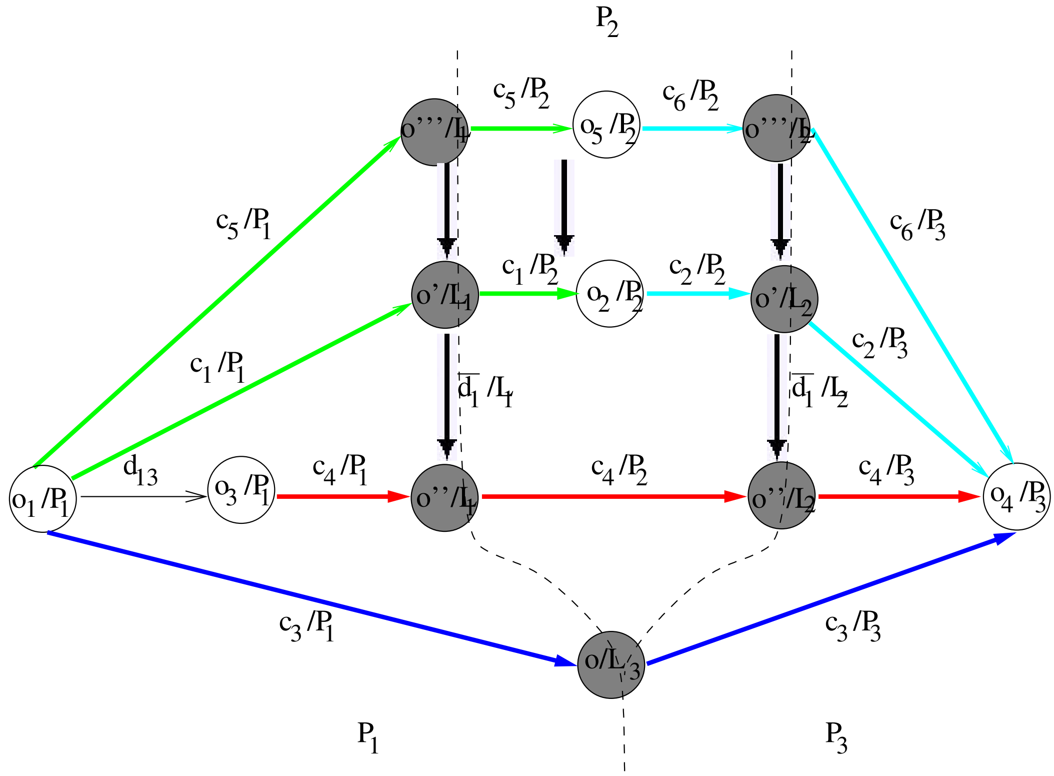
## Example 2/3




Distribution of operations and data dependences

# Formalization of the AAA implementation

## Example 3/3




Scheduling of computation and communication operations

# Optimized implementation: adequation

## Principles 1/3

Among a number of valid implementations that can be very large, we have to seek manually or automatically an **optimized implementation** called **adequation**.

The automatic optimization problem consists, first, in choosing among the valid implementations, the ones such that each operation satisfies **data dependences constraints** as well as **deadline equal to the period constraints**, and then in **minimizing the latency** of the algorithm implemented on the architecture. Moreover, for example, the number of components of the architecture could be minimized.

In order to choose among the valid implementations, every operation and data dependence of the algorithm graph must be characterized in term of **execution time** relatively to the architecture, and possibly in terms of **period** and **deadline equal to this period**. This leads to an algorithm graph labelled with these characteristics.

# Optimized implementation: adequation

## Principles 2/3

The **optimal** implementation problem is equivalent to a "Bin Packing" problem. It belongs to the **NP-hard** complexity class which contains problems more difficult than those of the **NP-complete** class, which contains the more difficult problems of the **NP** class. This latter contains problems that can be solved on a **N**on deterministic Turing machine by an algorithm in polynomial time relatively to the size of the problem. Thus, NP does not means "Non Polynomial". These problems can be solved by listing all the solutions, each of them being tested in polynomial time. The **P** class contains problems that can be solved with a deterministic Turing machine by an algorithm in **P**olynomial time. Deterministic (resp. non deterministic) means that from any state of the Turing machine, there is one (resp. several) possible transition. NP-hard problems are optimally solved in **exponential time**.

For small size problems, exact optimal algorithms can be used like linear, dynamic or constraint programming, branch and bound, branch and cut, etc.

# Optimized implementation: adequation

## Principles 3/3

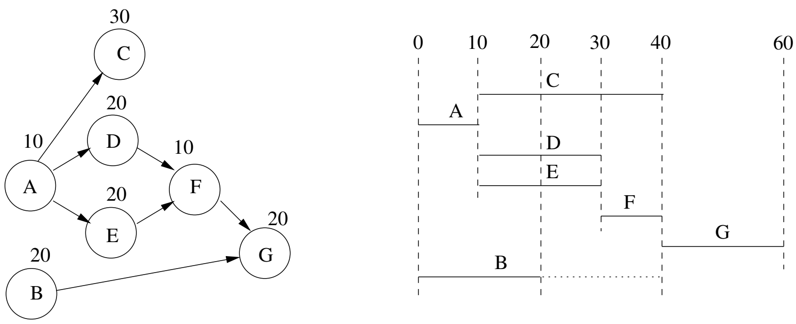
For realistic size problems, **heuristics** can be used that provide solutions which are empirically close to the optimal solution. **Approximation algorithms** find solutions close to the optimal one with a factor $\epsilon$.
There are two types of heuristics:

- without backtracking: very fast, called **greedy**. They do not start with an initial solution, and search at every step for a locally optimal solution to build a final solution which is not necessarily globally optimal. They perform deterministic choices, generally in a list. They are adapted to specific problems;
- with backtracking: slower, called **local search**. They start from an initial solution that they iteratively transform to improve it by searching in the **neighbouring** of the current solution. They can perform non deterministic choices to jump out from local minima. They give results that are empirically closer of the optimal solution than greedy heuristics. Since they solve generic problems they are called **metaheuristics**: simulated anealing, tabu search, genetic algorithm, ant colony, etc.

# Optimized implementation: adequation

## Critical path

The implementation while minimizing the latency is based on the computation of the **critical path** (CP) of the algorithm labelled only with execution times of the computation operations. Communication costs are not considered. A segment of length equal to its execution time, is associated to every operation. This segment is positioned at its **earliest start date** according to its predecessors allowing to determine its **latest start date** according to its successors. A CP corresponds to a path of segments without **schedule flexibility**, i.e. such that each segment has its earliest start date equal to its latest start date. The CP is equal to the Max of the CP when there exist several ones.




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

# Optimized implementation: adequation

## Operation and data dependence characterization

Every operation is **characterized relatively to the different operators** that are able to execute it. Every data dependence is **characterized relatively to the different media** that are able to execute it.

- **operator** and **communicator**
  operation name → execution time (without arbitration)
  data transfered → execution time (without arbitration)

| 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** (arbitration) operator1 is slowed down when operator2 is active

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

Input and output DMA are 50% slowed down because they access to a shared memory.

# Optimized implementation: adequation

## Heuristic minimizing the latency = input rate = strict period 1/4

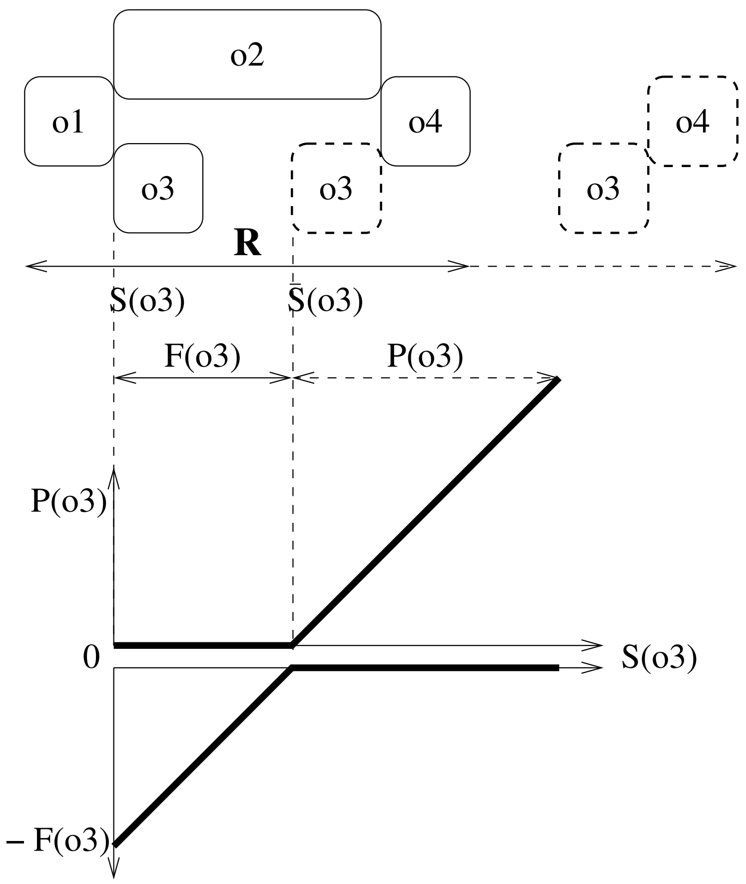
We use a **greedy heuristic** for a fast distribution and scheduling which is of low complexity $Card(\mathcal{O})Card(\mathcal{P})$.

In order to satify the partial order defined by the data dependences between operations of the algorithm graph, the heuristic chooses at step $i$ an operation in the subset of operations whose predecessors were already distributed and scheduled, called **candidates**, which optimizes a local cost function. The chosen operation is removed from the initial set of operations. The cost function, called **schedule pressure**, is defined by $\sigma(o, p) = P - F$ with:

- $F$ the difference between the earliest start date and the latest start date, called **schedule flexibility**,
- $P$ the lengthening of the **critical path** caused by the communication costs, called **schedule penalty**. It corresponds to a partial execution time or partial latency.

# Optimized implementation: adequation

## Heuristic minimizing the latency = input rate = strict period 2/4




o1 $\rightarrow$ (o2, o3) $\rightarrow$ o4

$S(o3)$ earliest start date of o3
$\overline{S}(o3)$ latest start date of o3
$E(o3) = S(o3) + C$ earliest end date
$F(o3)$ flexibility of o3
R partial latency

The earliest start date of o4 is moved back because of the duration of the communication due to o1 $\rightarrow$ o3 which is greater than the schedule flexibility $F$(o3) of o3, lengthening the partial latency R.

# Optimized implementation: adequation

## Heuristic minimizing the latency = input rate = strict period 3/4

**Candidates** $\mathcal{C}_i \subseteq \mathcal{O}$ at step $i$ are the subset of $\mathcal{O}$ whose predecessors were already distributed and scheduled.

MONO-PERIOD AAA HEURISTIC

1: $i = 0$, $V_0 = \mathcal{O}$ operations of the algorithm graph

**While** $V_i \neq \emptyset$

$i = i + 1$

**For all** $o_j \in \mathcal{C}_i \subset V_i$

**For all** $p_k \in P$ compute $\sigma(o_j, p_k)$

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

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

compute the partial latency, $V_i = V_{i-1} - \{o_j\}$

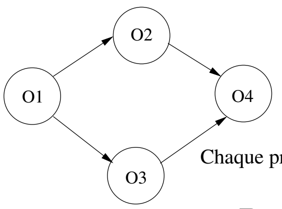
**End while**

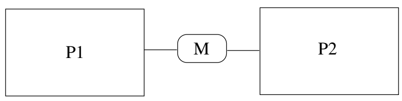
**If** latency (last partial latency) $\leq$ latency constraint **End**

**Else** modify distribution/scheduling constraints and/or increase the potential parallelism of the algorithm $\mathcal{O} = \mathcal{O}'$ **Go to** 1

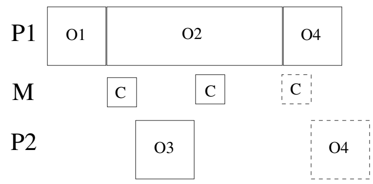
# Optimized implementation: adequation

## Heuristic minimizing the latency = input rate = strict period 4/4




Chaque processeur contient un opérateur et un communicateur




Execution time P1 or P2: o1=10, o2=30, o3=10, o4=10, M: integer=5

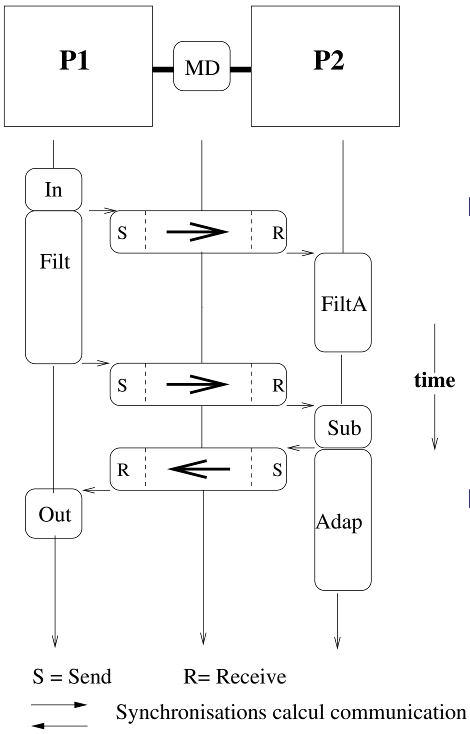
| i | $(o,p)$ | $\sigma(o,p)$ | min $\sigma$ | max(min $\sigma$) | $V_0 = \{o1, o2, o3, o4\}$ |
|---|---|---|---|---|---|
| 1 | (o1,P1) | 10 | (o1,P1) | (o1,P1) | $V_1 = \{o2, o3, o4\}$ |
| 2 | (o2,P1)<br>(o2,P2)<br>(o3,P1)<br>(o3,P2) | 10+30=40<br>(10+5)+30=45<br>10+10=20<br>(10+5)+10=25 | (o2,P1)<br>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

Uniprocessor 60, CP = 50, latency = 50, maximal acceleration without comm. = 60/50 = 1.2, acceleration with comm. = 60/50 = 1.2

# Optimized implementation: adequation

## Adequation result: scheduling table

Equalizer implemented on two processors, each containing an **operator** and one **communicator** in parallel and a point-to-point medium.




- The result of the adequation is an **implentation graph** whose partial order is compatible with the initial partial order of the algorithm. It allows the production of a **scheduling table**.
- Every communication operation composed of a send S and a reception R of a data message, is executed on processors P1 and P2 by the communicators. Identically, for write and read of a shared memory.
- For every processor and medium of the architecture graph, the implementation graph gives **start dates** and **end dates** of the computation and communication operations. Their length is proportional to their duration.

# Optimized implementation: adequation

## Strict multi-period heuristic minimizing the latency

To every operation $o_i$ of the algorithm is associated a **strict period** $T_i$, independent of the processor $P_j$ and an execution time $C_i$ dependent of the processor $P_j$. The LCM of the periods $T_i$ is called **hyperperiod**.

`MULTI-PERIOD AAA HEURISTIC`

1 `Assignment`: **For all** operations $o_i$
  **For all** processors $P_j$
    **If** $o_{iP_j}$ schedulable **Then** $j + 1$ **Else End**
  **(sufficient feasibility condition: $\sum_{i=1}^{n} C_{iP_j} \leq GCD(T_i)$)**

2 `Unrolling`: **For all** operations, duplicate it as many times as the ratio between the hyperperiod and the period of the operation and add the necessary edges

3 `Scheduling`: **For all** operations assigned and unrolled on a processor, compute its earlieast start date while taking into account inter-processor communication costs
**If** an operation was assigned to several processors
**Then** choose the one minimizing the latency

## Optimized implementation: adequation

### Heuristic minimizing the input rate, acceleration

**Heuristic minimizing the input rate**

- Search for critical cycles
- Retiming associated to the delays
- Increase the latency

**Acceleration for an homogeneous architecture**

$$\textbf{Maximal acceleration} = \frac{\text{sum of all the execution times of operations}}{\text{critical path duration without communications}}$$

$\lceil \text{Maximal acceleration} \rceil = [Card(\mathcal{P})] =$ Maximal number of identical processors put in **actual parallelism** necessary to exploit the **potential parallelism** while taking into account execution times.

processors in actual parallelism $\leq$ processors in potential parallelism

## Optimized implementation: adequation

### Heuristic for minimizing the number of processors

We can minimize the number of processors initially given. We use a **meta-heuristic** (different from **metaheuristic**) which calls a heuristic.

`AAA META-HEURISTIC`

$nbProc$ = Initial number of processors = $[Card(\mathcal{P})]$
Call `AAA HEURISTIQUE`
**While** the latency constraint is satified
    $nbProc = nbProc - 1$
    Call `AAA HEURISTIQUE`
**EndWhile**

# Code generation

## General issues 1/2

Since real-time systems are specified according to a LTT approach, we **favour implementations on TT architectures** (periodic polling of sensors)rather than ET architectures (interruptions provided by sensors). The polling of the sensors, the execution of one infinite repetition of the algorithm, and the writing on the actuators, can be triggered by:

- a periodic time base,
- a loop of known duration, called **auto-triggered**.

**From now on we assume to be in this latter case.**

On every processor the **executive** code is **offline** if optimization choices and decisions are taken before the execution of the system and **online** if the choices and decisions are taken during the execution. For hard real-time systems the **offline approach is favoured** since it is consistent with the TT approach.

In this case the executive reads the **scheduling table** containing the sequence of operations, of waiting operations and of communication operations, to infinitely repeat.

# Code generation

## General issues 2/2

- The **executive code** is composed of system instructions which control (scheduling, conditioning, repetition) the **applicative code** associated to every operation specified in the algorithm.
- The executive code is automatically **custom synthetized** according to the application. It benefits at best from application characteristics and induces a low overhead easy to determine and, as such, deterministic.
- The custom synthetized executive may call a **resident executive** code which is generic and partially benefits from application characteristics. The resident executive code is generally dynamic involving a higher overhead, that is difficult to determine but has the advantage to be "standard".
- The cost of the executive code must be taken into account as precisely as possible, so that the schedulability analysis is reliable.
- For every processor a **pseudo code** is automatically generated such that it is **architecture independent**. It is called **macro-code** and composed of a macro-executive and applicative macros.

# Code generation

## Macro-code generation from adequation result 1/2

Every **macro-code** is composed of an **infinite loop** sequencing:

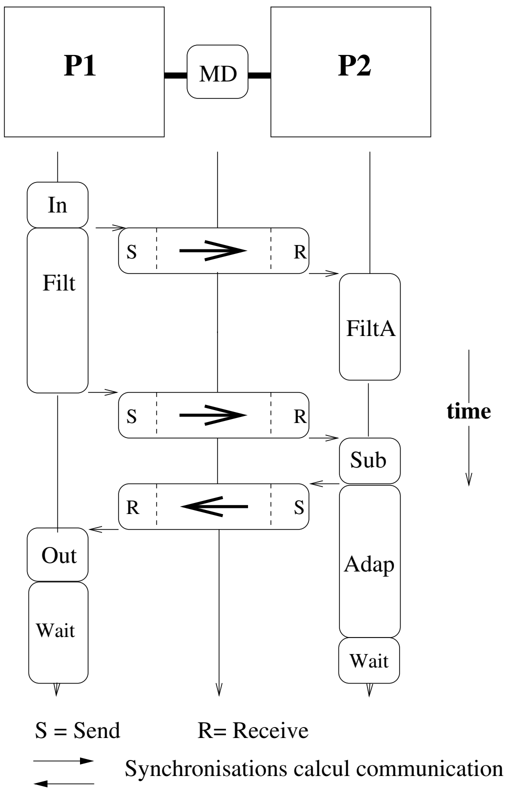
- **system macros** composing the **macro-executive**:
  - **conditioning**: choice of operations according to a condition,
  - **wainting (`wait`)**: delay after some operation to guarantee its period,
  - inter-processor **communication**: for the SAM send a data message from a communicator (`send`) and reception of this message by a communicator (`recv`), and for the RAM write of a data (`write`) and read (`read`) of this data,
  - **synchronization** *intra-processor* and *inter-processor*.
- **applicative macros** actually performing **operations** distributed and scheduled on this processor, a buffer is associated to each output.

Synchronizations guarantee that even if there are **variations on operation execution times** their partial order of execution will be compatible with the initial partial order of the algorithm graph. Such designs are called latency insensible (LID).

# Code generation

## Macro-code generation from adequation result 2/2

TT architecture composed of 2 processors `P1` and `P2`, each containing an operator and one communicator in parallel and a point-to-point medium.




**Operator of P1**: macro-loop of tasks, macros (`In`, `Filt`, `Out`, `Wait`).
**Communicator of P1**: macro-loop of communications, macros (`send`, `send` et `recv`).
**Operator of P2**: macro-loop of tasks, macros (`FiltA`, `Sub`, `Adap`, `Wait`).
**Communicateur de P2**: macro-loop of communications, macros (`recv`, `recv`, `send`).
Every task loop of an operator and every communication loop of a communicator, contains additional macros for synchronizing operator and communicator.
The **wait** macros garantee an auto-triggering, at a given period, greater than the optimized latency.

# Code generation

## Synchronizations in the macro-code: principles 1/2

**Intra-processor synchronizations** of two types:

- intra-repetition: to guarantee the parallel execution, **correct according to the initial partial order**, of the unique computation sequence and of the communication sequences, inside an infinite repetition,
- inter-repetition: to guarantee that infinite repetitions **correctly succeed each other**. An infinite repetition must be completed before the next one starts, thus every sent message must have been received before sending the next one, using the corresponding SAM, or that every data written in the shared memory must have been read, using the corresponding RAM.

Synchronization macros atomically perform a **read-modify-write** in a semaphore as follows:

- intra-repetition: `Pre_full`, `Succ_full`: signal full buffer, wait buffer full,
- inter-repetition: `Pre_empty`, `Succ_empty`: signal empty buffer, wait empty buffer.

# Code generation

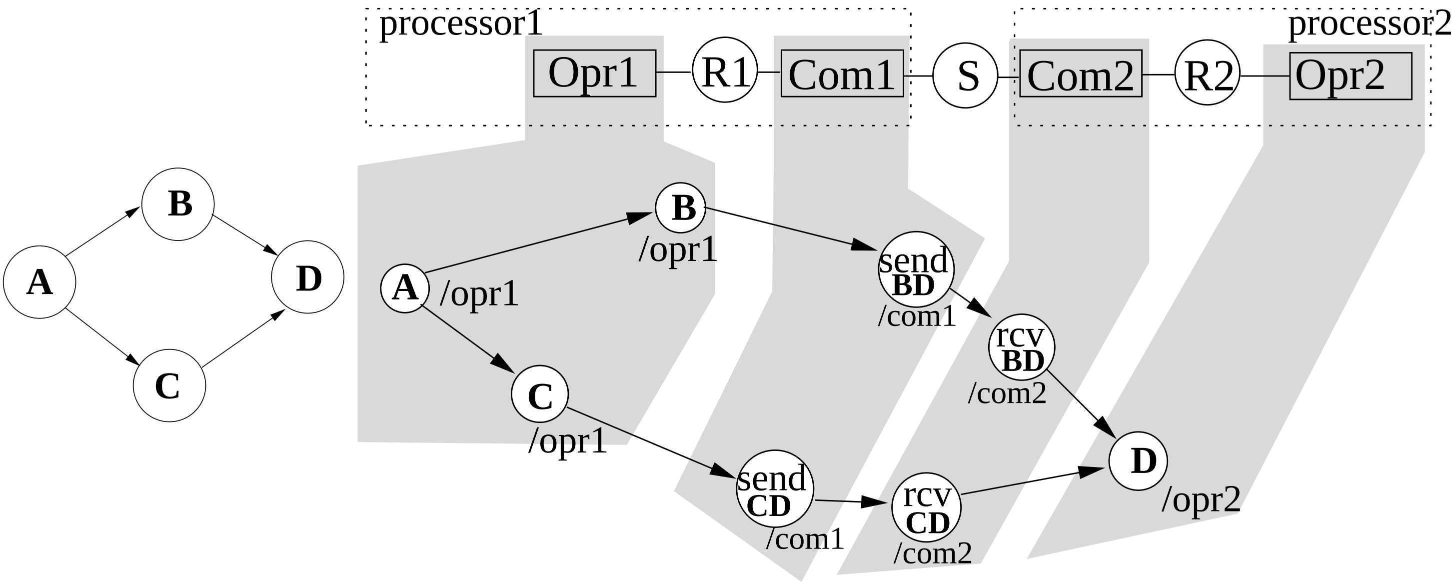
## Synchronizations in the macro-code: principles 2/2

**Inter-processor synchronizations** perform synchronizations between communicators macro-loops of several processors, using for:

- message passing:
  - point-to-point medium: no synchronization message since the FIFO already performs the synchronisation,
  - multi-point diffusing medium: `send` the data to all the processors, received with a `sync` by the processor which does not use it,
  - multi-point non diffusing medium: send synchronization messages `send_synchro` and receive synchronization messages `recv_synchro`, for the corresponding processors,
- shared memory an additional semaphore:
  - `PreR_full`, `SuccR_full`: signal full memory, wait full memory,
  - `PreR_empty`, `SuccR_empty`: signal empty memory, signal empty memory.

# Code generation

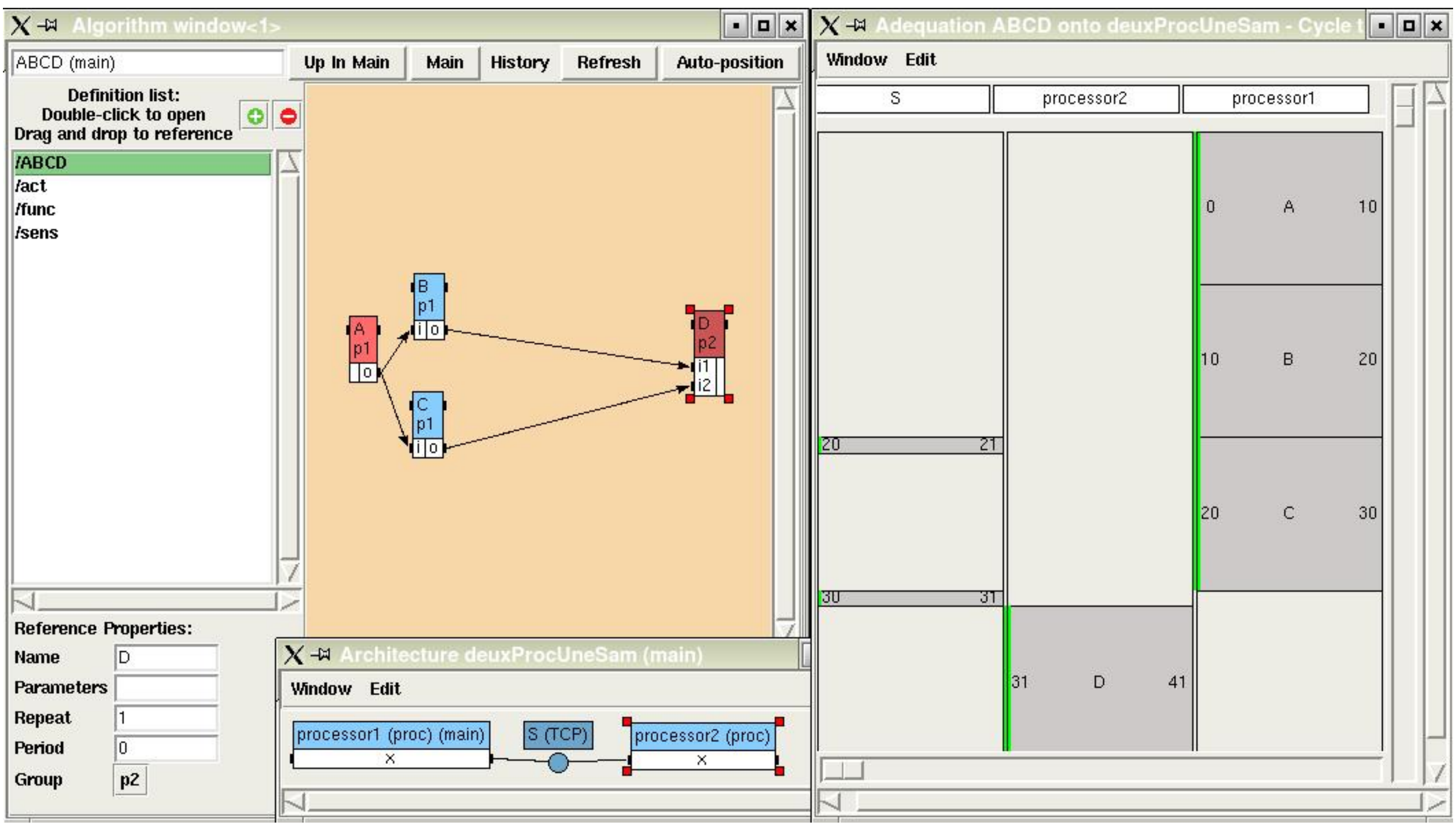
Synchronizations: ABCD algorithm and architecture graphs, scheduling table




R1 and R2 memories contain buffers in which operations B and C (resp. D) produce (resp. consume) their data.

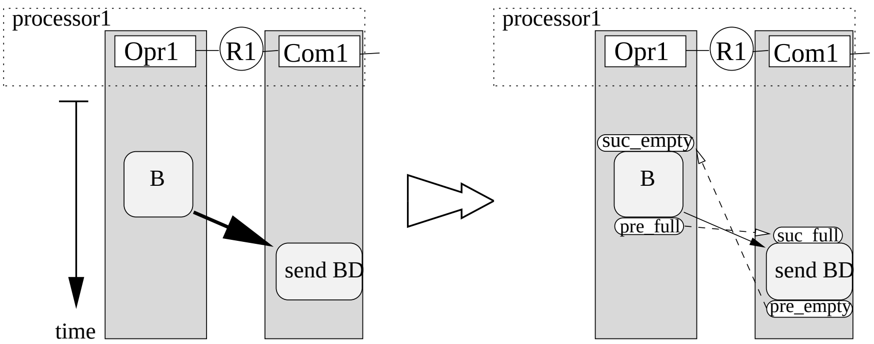
# Code generation

Intra-processor synchronizations: adequation, point-to point message passing communication, ABCD algorithm

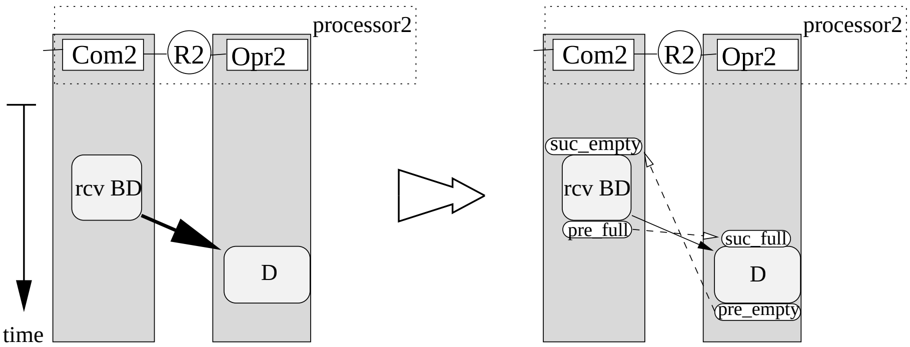


# Code generation

Intra-processor synchronizations: point-to point message passing communication, `B-Send` and `Receive-D`



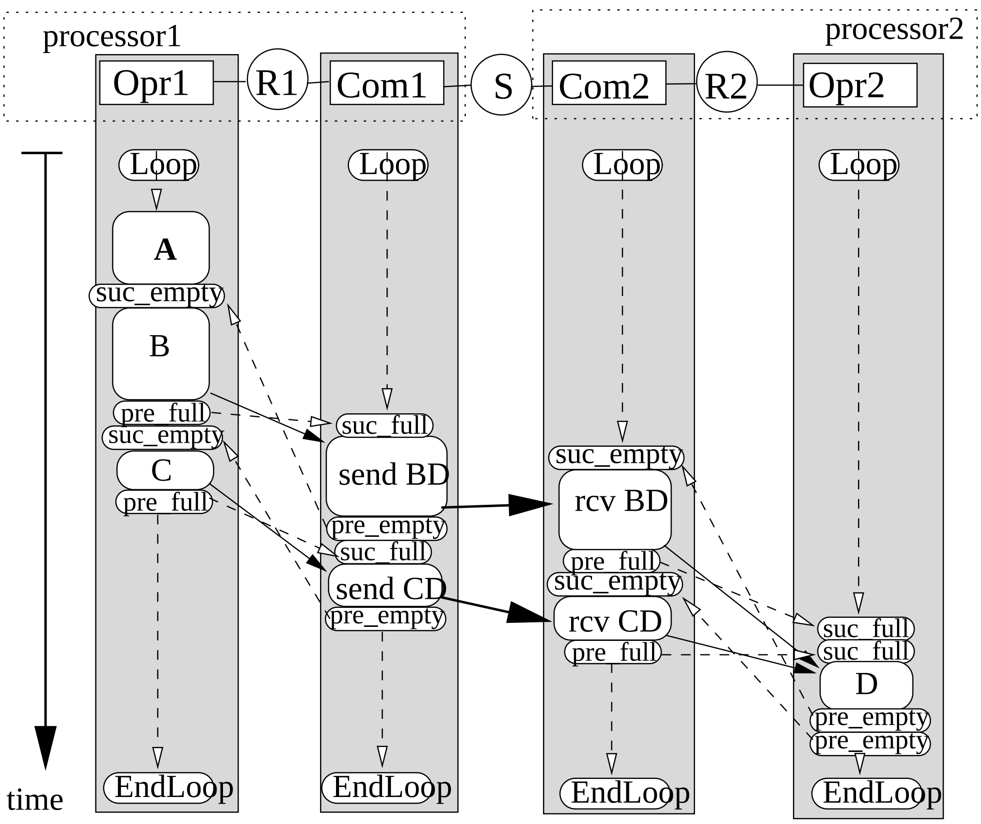

**Intra-repetition synchro.** (Pre_full, Suc_full): buffer full before execution of `send`.
**Inter-repetition synchro.** (Suc_empty, Pre_empty): buffer empty before sent of data.




**Intra-repetition synchro.** (Pre_full, Suc_full): buffer full before execution of `recv`.
**Inter-repetition synchro.** (Suc_empty, Pre_empty): buffer empty before reception of data.

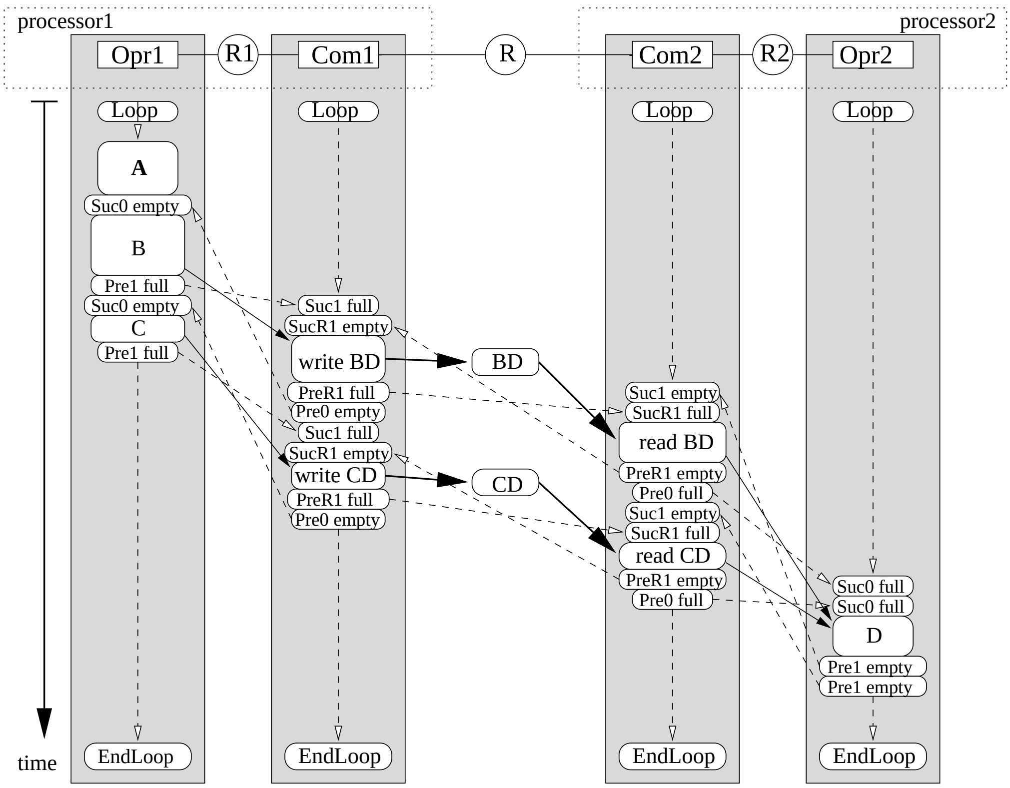
# Code generation

Intra-processor synchronizations: point-to point message passing communication, ABCD algo



# Code generation

## Intra-processor synchronizations: shared memory communication, algorithme ABCD




# Code generation

## Macro-code generation: intra-processeur synchronization

## Point-to-point message passing communication, ABCD algorithm, 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)
```

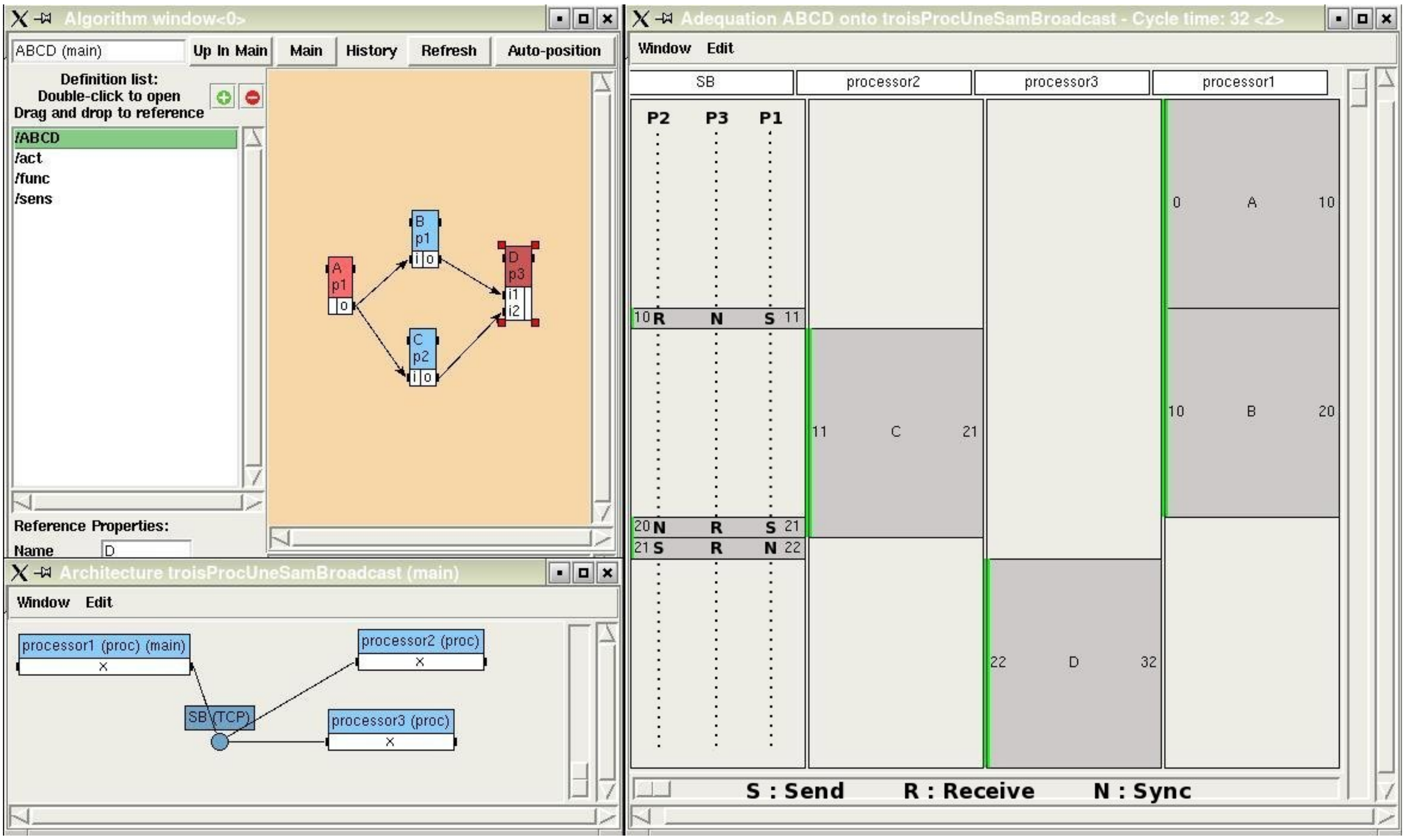
```
main_
  spawn_thread_(x)
  sens(_ABCD_A_o)
  loop_
    sens(_ABCD_A_o)
    Suc0_(_ABCD_B_o_processor1_x_empty,x,_ABCD_B_o,empty)
    func(_ABCD_A_o,_ABCD_B_o)
    Pre1_(_ABCD_B_o_processor1_x_full,x,_ABCD_B_o,full)
    Suc0_(_ABCD_C_o_processor1_x_empty,x,_ABCD_C_o,empty)
    func(_ABCD_A_o,_ABCD_C_o)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

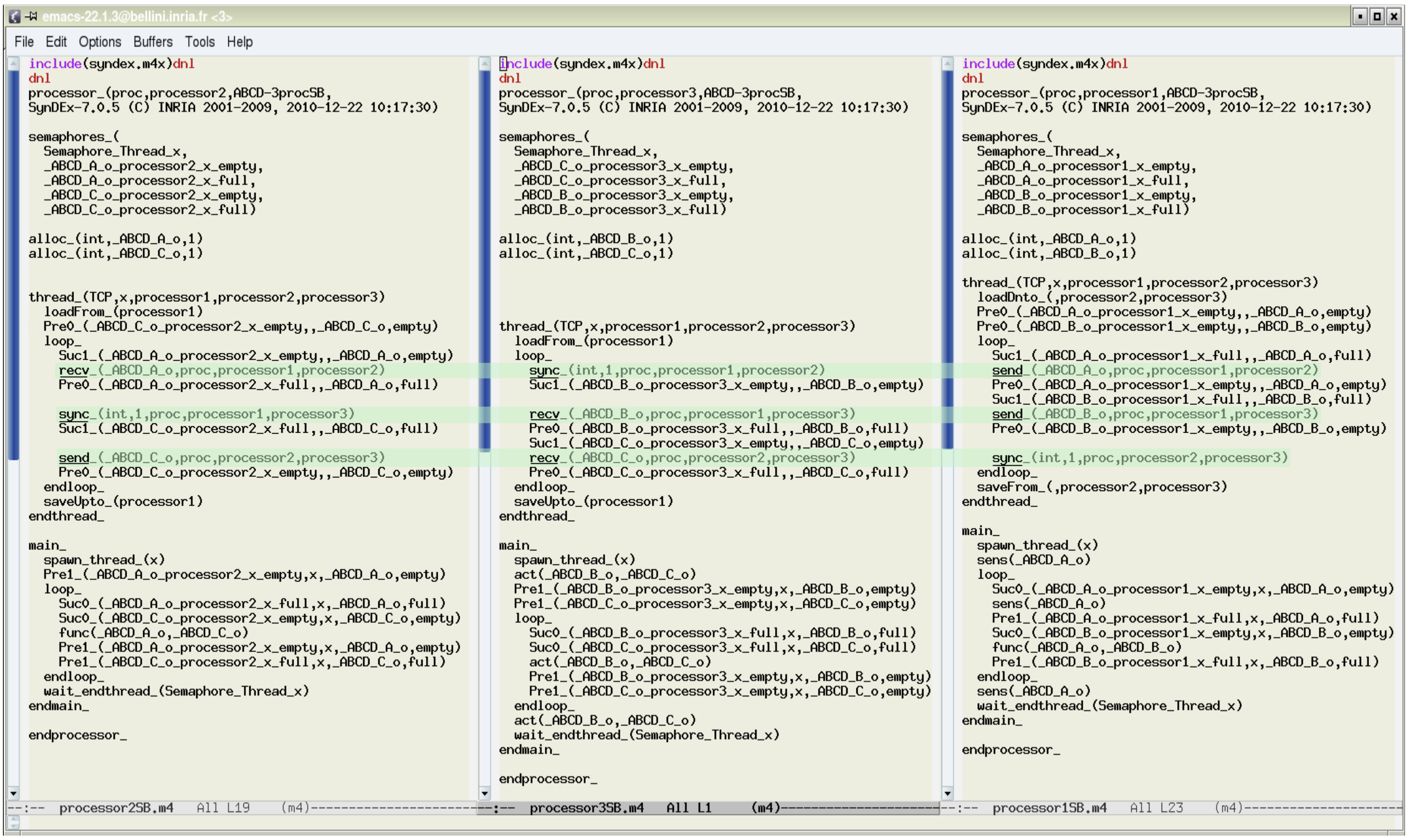
# Code generation

## Inter-processor synchronizations: adequation, diffusing multi-point message passing communication, ABCD algorithm, 3 processors




# Code generation

## Macro-code generation: Inter-processor synchronizations diffusing multi-point message passing communication, ABCD algorithm, 3 processors



```
include(syndex.m4x)dnl
dnl
processor_(proc,processor2,ABCD-3procSB,
SynDEx-7.0.5 (C) INRIA 2001-2009, 2010-12-22 10:17:30)

semaphores_(
  Semaphore_Thread_x,
  _ABCD_A_o_processor2_x_empty,
  _ABCD_A_o_processor2_x_full,
  _ABCD_C_o_processor2_x_empty,
  _ABCD_C_o_processor2_x_full)

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


thread_(TCP,x,processor1,processor2,processor3)
  loadFrom_(processor1)
  Pre0_(_ABCD_C_o_processor2_x_empty,,_ABCD_C_o,empty)
  loop_
    Suc1_(_ABCD_A_o_processor2_x_empty,,_ABCD_A_o,empty)
    recv_(_ABCD_A_o,proc,processor1,processor2)
    Pre0_(_ABCD_A_o_processor2_x_full,,_ABCD_A_o,full)

    sync_(int,1,proc,processor1,processor3)
    Suc1_(_ABCD_C_o_processor2_x_full,,_ABCD_C_o,full)

    send_(_ABCD_C_o,proc,processor2,processor3)
    Pre0_(_ABCD_C_o_processor2_x_empty,,_ABCD_C_o,empty)
  endloop_
  saveUpto_(processor1)
endthread_

main_
  spawn_thread_(x)
  Pre1_(_ABCD_A_o_processor2_x_empty,x,_ABCD_A_o,empty)
  loop_
    Suc0_(_ABCD_A_o_processor2_x_full,x,_ABCD_A_o,full)
    Suc0_(_ABCD_C_o_processor2_x_empty,x,_ABCD_C_o,empty)
    func(_ABCD_A_o,_ABCD_C_o)
    Pre1_(_ABCD_A_o_processor2_x_empty,x,_ABCD_A_o,empty)
    Pre1_(_ABCD_C_o_processor2_x_full,x,_ABCD_C_o,full)
  endloop_
  wait_endthread_(Semaphore_Thread_x)
endmain_

endprocessor_
```

--:-- processor2SB.m4 All L19 (m4)

```
include(syndex.m4x)dnl
dnl
processor_(proc,processor3,ABCD-3procSB,
SynDEx-7.0.5 (C) INRIA 2001-2009, 2010-12-22 10:17:30)

semaphores_(
  Semaphore_Thread_x,
  _ABCD_C_o_processor3_x_empty,
  _ABCD_C_o_processor3_x_full,
  _ABCD_B_o_processor3_x_empty,
  _ABCD_B_o_processor3_x_full)

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



thread_(TCP,x,processor1,processor2,processor3)
  loadFrom_(processor1)
  loop_
    sync_(int,1,proc,processor1,processor2)
    Suc1_(_ABCD_B_o_processor3_x_empty,,_ABCD_B_o,empty)

    recv_(_ABCD_B_o,proc,processor1,processor3)
    Pre0_(_ABCD_B_o_processor3_x_full,,_ABCD_B_o,full)
    Suc1_(_ABCD_C_o_processor3_x_empty,,_ABCD_C_o,empty)
    recv_(_ABCD_C_o,proc,processor2,processor3)
    Pre0_(_ABCD_C_o_processor3_x_full,,_ABCD_C_o,full)
  endloop_
  saveUpto_(processor1)
endthread_

main_
  spawn_thread_(x)
  act(_ABCD_B_o,_ABCD_C_o)
  Pre1_(_ABCD_B_o_processor3_x_empty,x,_ABCD_B_o,empty)
  Pre1_(_ABCD_C_o_processor3_x_empty,x,_ABCD_C_o,empty)
  loop_
    Suc0_(_ABCD_B_o_processor3_x_full,x,_ABCD_B_o,full)
    Suc0_(_ABCD_C_o_processor3_x_full,x,_ABCD_C_o,full)
    act(_ABCD_B_o,_ABCD_C_o)
    Pre1_(_ABCD_B_o_processor3_x_empty,x,_ABCD_B_o,empty)
    Pre1_(_ABCD_C_o_processor3_x_empty,x,_ABCD_C_o,empty)
  endloop_
  act(_ABCD_B_o,_ABCD_C_o)
  wait_endthread_(Semaphore_Thread_x)
endmain_

endprocessor_
```

--:-- processor3SB.m4 All L1 (m4)

```
include(syndex.m4x)dnl
dnl
processor_(proc,processor1,ABCD-3procSB,
SynDEx-7.0.5 (C) INRIA 2001-2009, 2010-12-22 10:17:30)

semaphores_(
  Semaphore_Thread_x,
  _ABCD_A_o_processor1_x_empty,
  _ABCD_A_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)

thread_(TCP,x,processor1,processor2,processor3)
  loadDnto_(,processor2,processor3)
  Pre0_(_ABCD_A_o_processor1_x_empty,,_ABCD_A_o,empty)
  Pre0_(_ABCD_B_o_processor1_x_empty,,_ABCD_B_o,empty)
  loop_
    Suc1_(_ABCD_A_o_processor1_x_full,,_ABCD_A_o,full)
    send_(_ABCD_A_o,proc,processor1,processor2)
    Pre0_(_ABCD_A_o_processor1_x_empty,,_ABCD_A_o,empty)
    Suc1_(_ABCD_B_o_processor1_x_full,,_ABCD_B_o,full)
    send_(_ABCD_B_o,proc,processor1,processor3)
    Pre0_(_ABCD_B_o_processor1_x_empty,,_ABCD_B_o,empty)

    sync_(int,1,proc,processor2,processor3)
  endloop_
  saveFrom_(,processor2,processor3)
endthread_

main_
  spawn_thread_(x)
  sens(_ABCD_A_o)
  loop_
    Suc0_(_ABCD_A_o_processor1_x_empty,x,_ABCD_A_o,empty)
    sens(_ABCD_A_o)
    Pre1_(_ABCD_A_o_processor1_x_full,x,_ABCD_A_o,full)
    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)
  endloop_
  sens(_ABCD_A_o)
  wait_endthread_(Semaphore_Thread_x)
endmain_

endprocessor_
```

--:-- processor1SB.m4 All L23 (m4)

# Code generation

## Distributed embedded real-time executable code generation 1/2

The macro-code of every processor is macro-processed with:

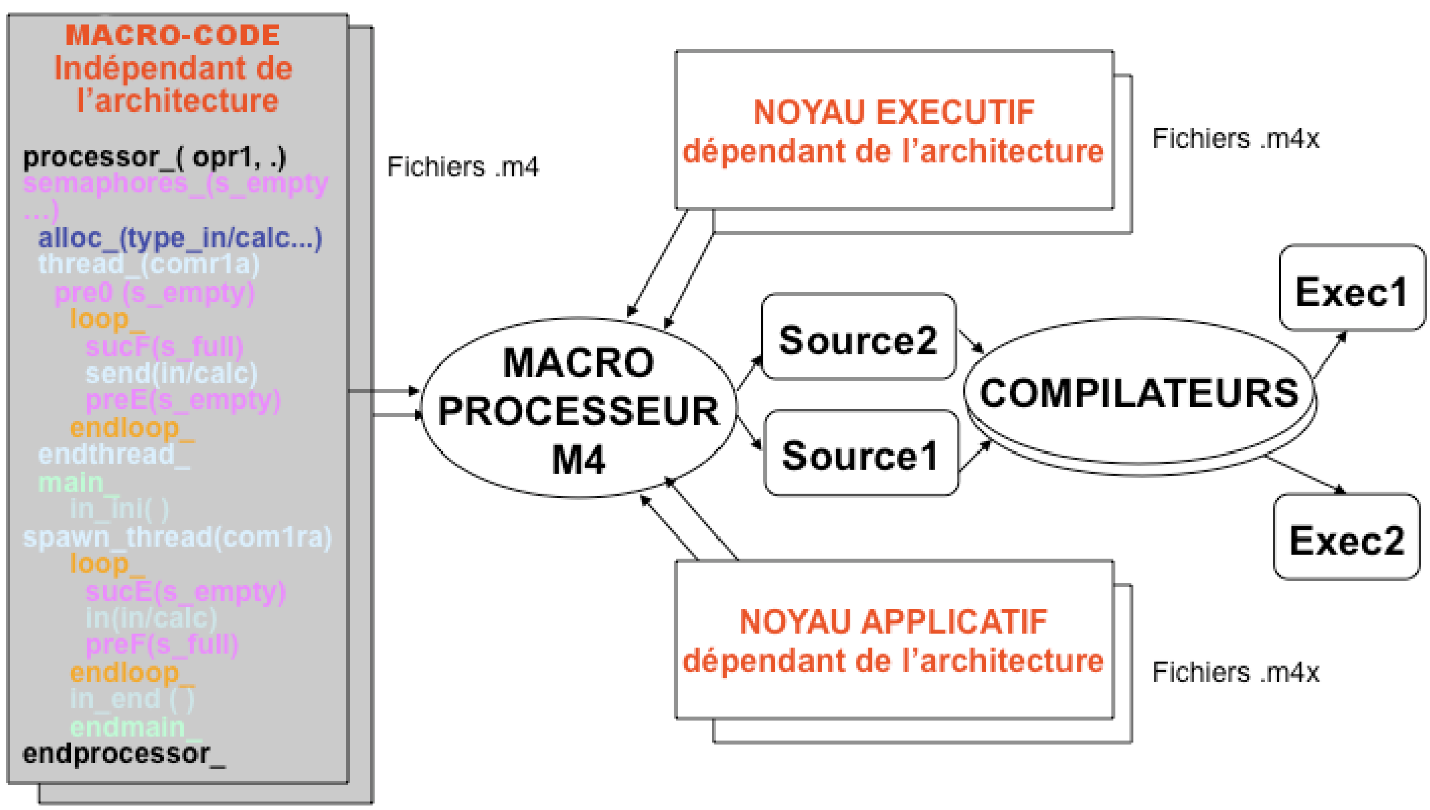
- an **executive kernel** which is **architecture independent** and possibly a **resident executive**, for example VxWorks, Osek, Linux/RTAI, Linux/Xenomai, Windows/RTX, etc. Every kernel contains the information describing how each macro-instruction will be translated in a compilable source code;
- an **applicative kernel** which is **architecture dependent** containing the macro-operations corresponding to the operations of the algorithm graph. Every applicative kernel contains the informations describing how each macro-operation will be translated in a compilable source code.

**The source codes are compiled to produce the executable codes**.

The obtained executable codes are such that synchronizations guarantee that the initial partial order of the algorithm graph is preserved during the automatic code generation producing a real-time execution **without any deadlock**.

# Code generation

## Distributed embedded real-time executable code generation 2/2



# SynDEx software

## Features 1/2

**SynDEx implements the AAA methodology**

**SynDEx** is an interactive graphical software which provides aids for the implementation of signal and image processing with control on multicomponent architectures while taking into account real-time constraints. It offers the following features:

- functional specifications,
  - algorithm graph spcification with a proprietary language,
  - interface with domain specific languages (DSL): Synchronous languages (ESTEREL/SYNCCHARTS, SIGNAL) (formal verifications and real-time simulation), SCICOS (modelling/simulation of hybrid systems), UML2/MARTE (UML profile OMG standard for embedded real-time), etc., producing this proprietary language,
- non functional specifications,
  - multicomponent graph specification,
  - timing characterization,

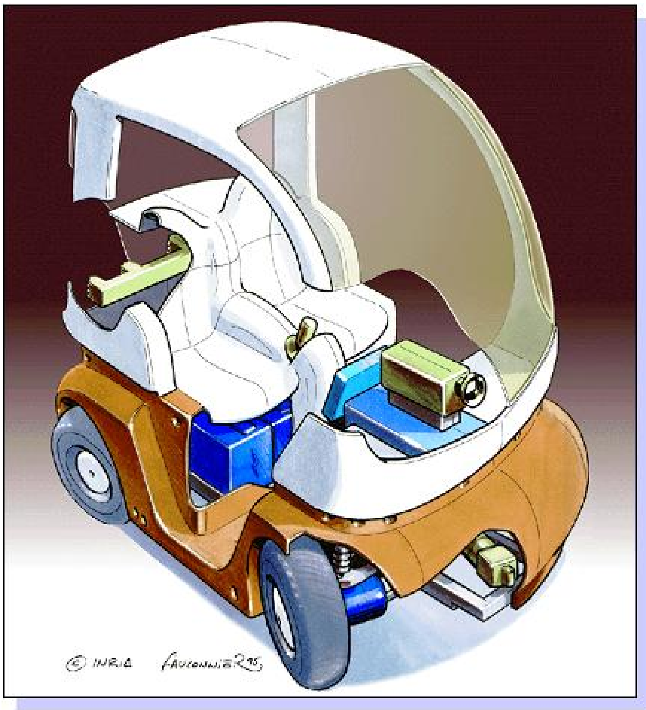
# SynDEx software

## Features 2/2

- adéquation,
  - distributed real-time schedulability analysis producing a scheduling table,
  - optimizations and choice of an implementation that preserves the properties of the functional specification,
  - vizualisation of a timing diagram of the distributed execution giving simulated performance measures,
- automatic generation of distributed real-time executives without deadlock that are custom synthetized from **executive kernels**:
  - for processors Analog Device ADSP21060, Texas Instrument TMS320C40, Microchip PIC182680, Intel ix86, i8051, i80C196, Motorola MC68332, MPC555, Transputer T80x,
  - possibly calling resident executives: Linux, Linux/RTAI, Linux/Xenomai, Windows Windows/RTX working stations Intel ix86 communicating with TCP/IP,
  - real-time performances measures with software probes introduced automatically during the automatic generation of executives.

# SynDEx software

## CyCab example 1/2

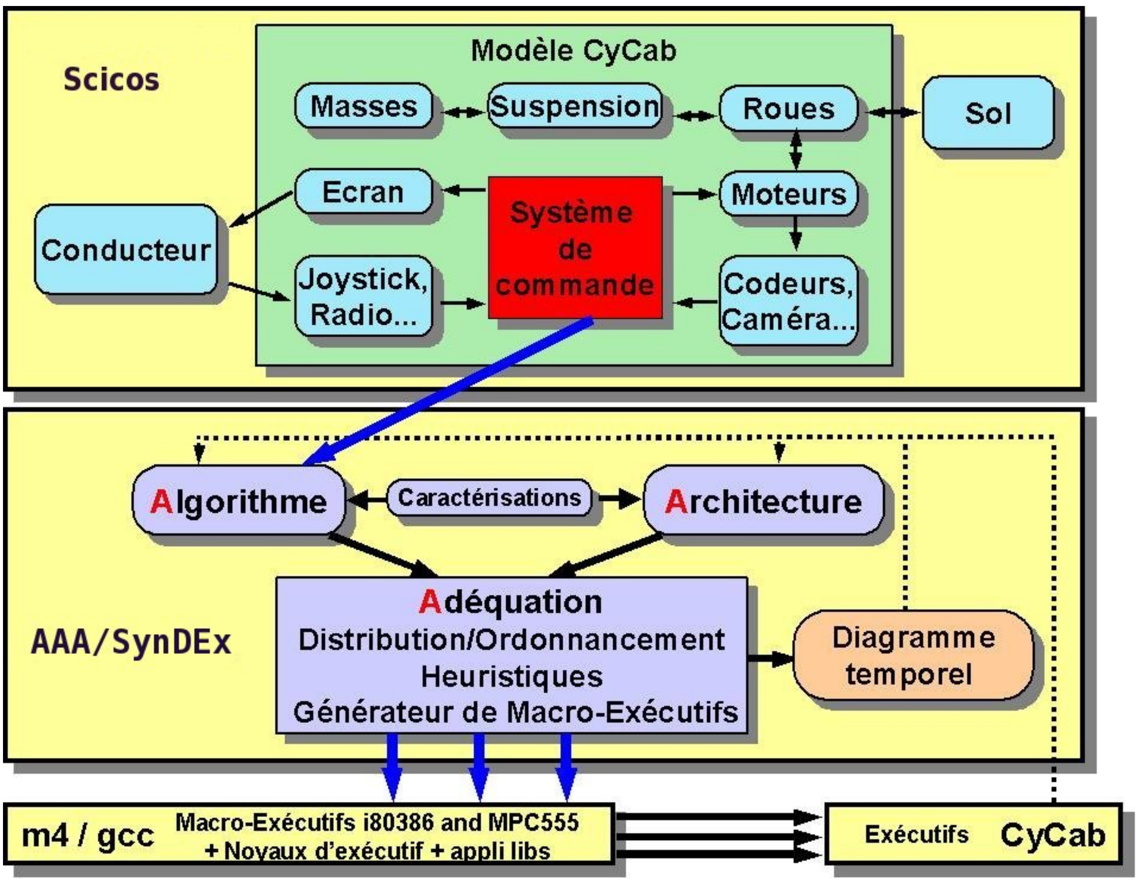



- Vitesse 30km/h
- Moteurs électriques
- 4 roues motrices
- 2 directions AV, AR
- Multi-processeur MPC555 + un PC embarqué
- Bus Can

Industrialisé par Robosoft
www.robosoft.fr

# Logiciel SynDEx

## CyCab example 2/2



# SynDEx software

## Utilization

The user **specifies with the graphical user interface** algorithm and architecture graphs, strict periods, WCET of operations and WCCT of data dependences, or he **imports** them through a **.sdx file** produced by the compiler of some DSL.

The AAA heuristic performs the schedulability analysis and computes the start time of every computation operation and of every communication operation from their period, WCET and WCCT. Then, the code generator produces as many macro-code files as there are of processors in the architecture.

The WCCT $d$ of a message passing communication operation, executed by the two communicators (`send, recv`) of two processors, is computed from a simple model, for example: $d = \tau + \delta * n$, where $\delta$ denotes the elementary WCCT for transfering one data element, $n$ the number of data elements (depending of the data type), $\tau$ the time necessary to establish the communication.

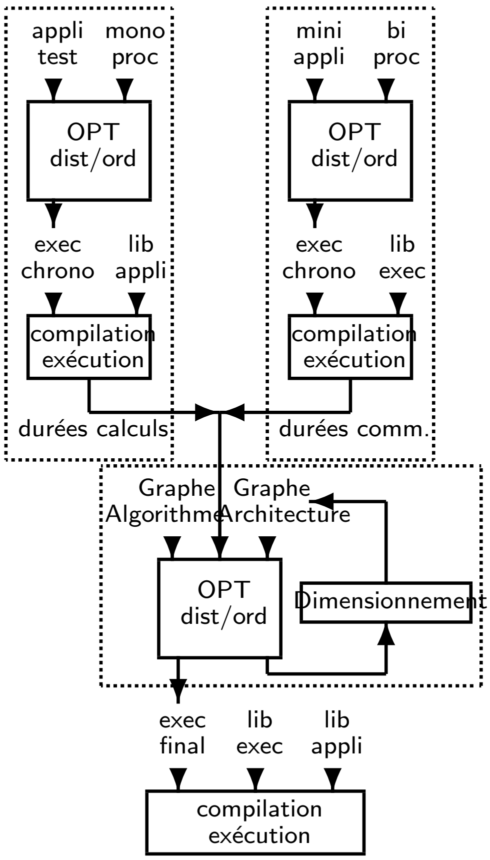
# SynDEx software

## Execution time measures

Using the option **"Code generation with timers"** during the code generation, adds for every computation or communication operation a first timer macro-code which gives the date before the operation execution and a second timer macro-code which gives the date after its execution.

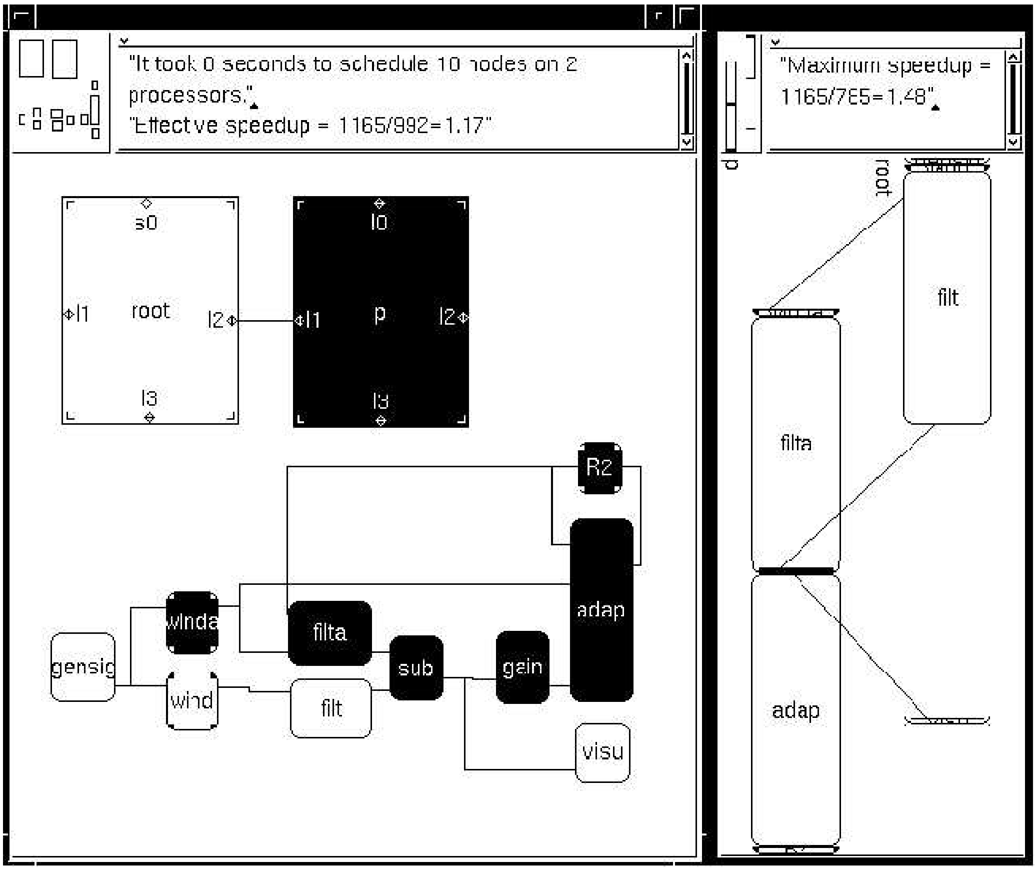
The considered algorithm application is executed on a uniprocessor architecture and the difference between the two previous dates gives the execution time of each computation operation.

A minimal application A $\rightarrow$ B executed on two processors, connected by the different possible media, gives the elementary WCCT of each communication operation.



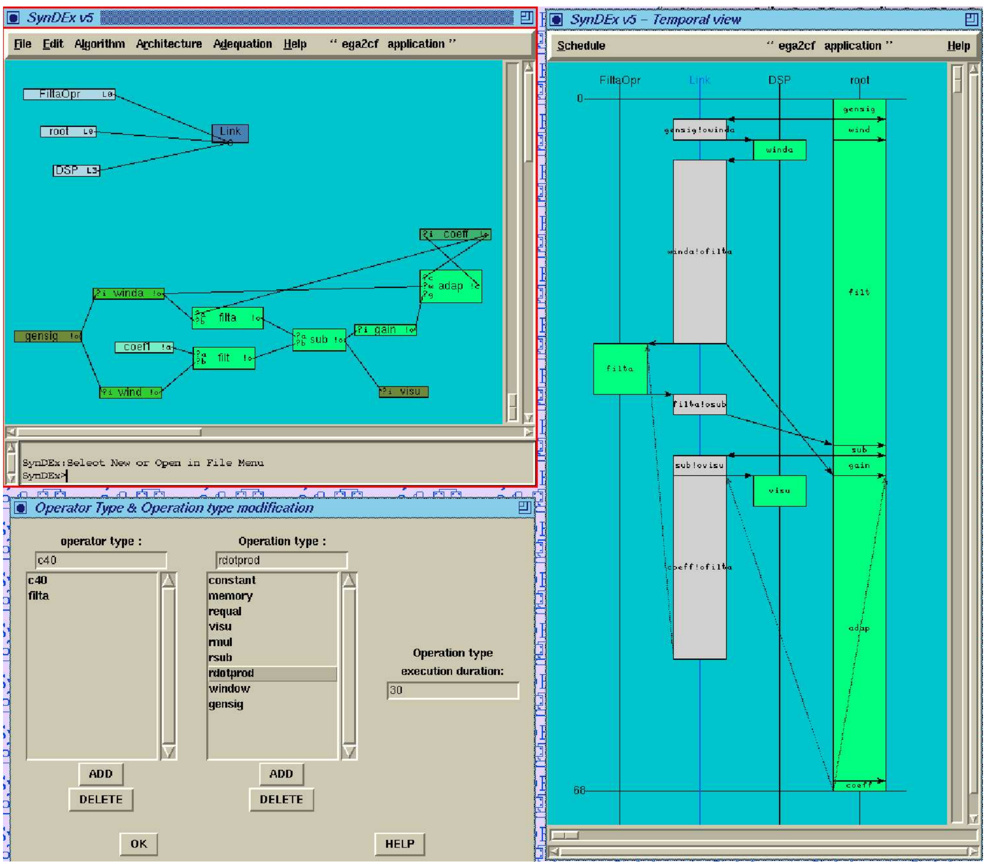
# SynDEx software

## GUI V4



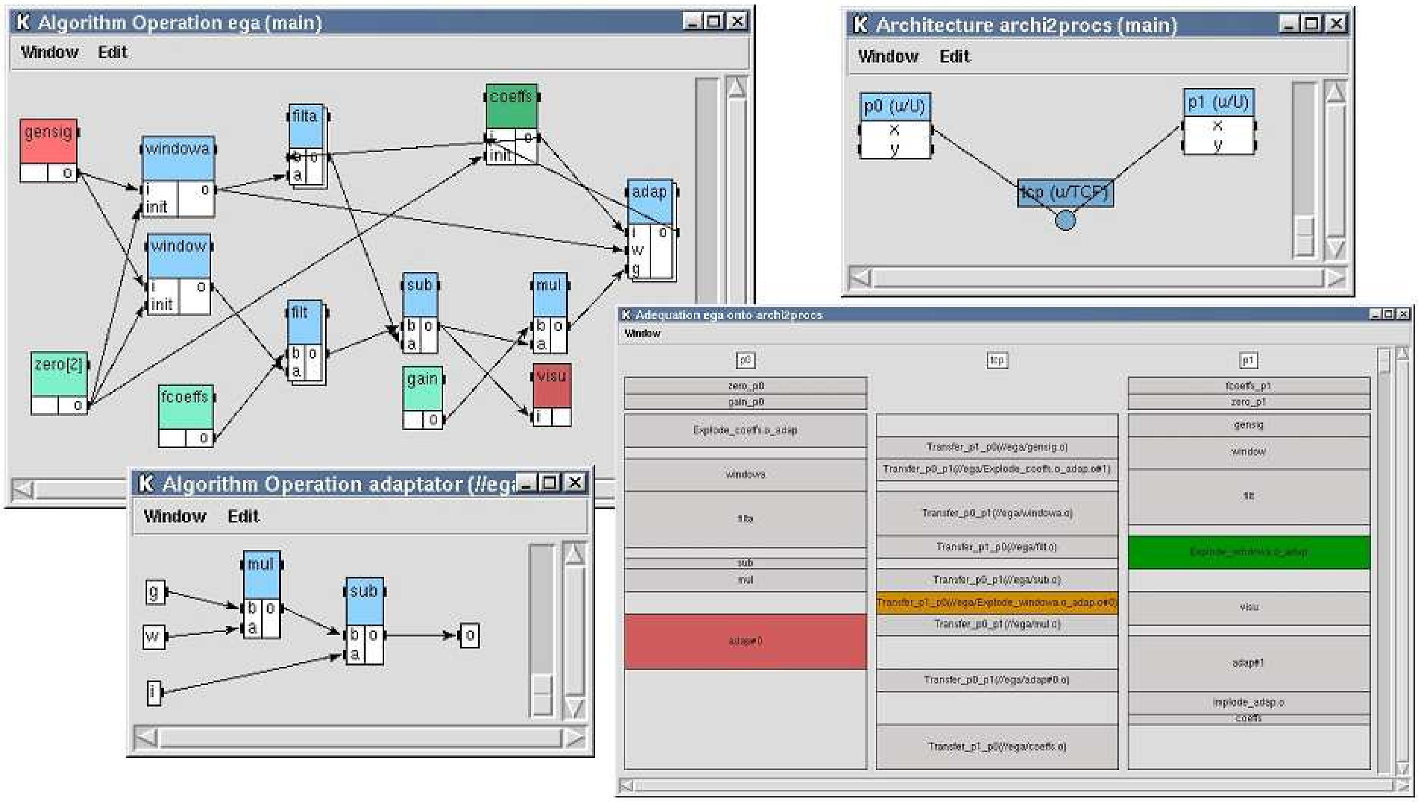

# SynDEx software

## GUI V5



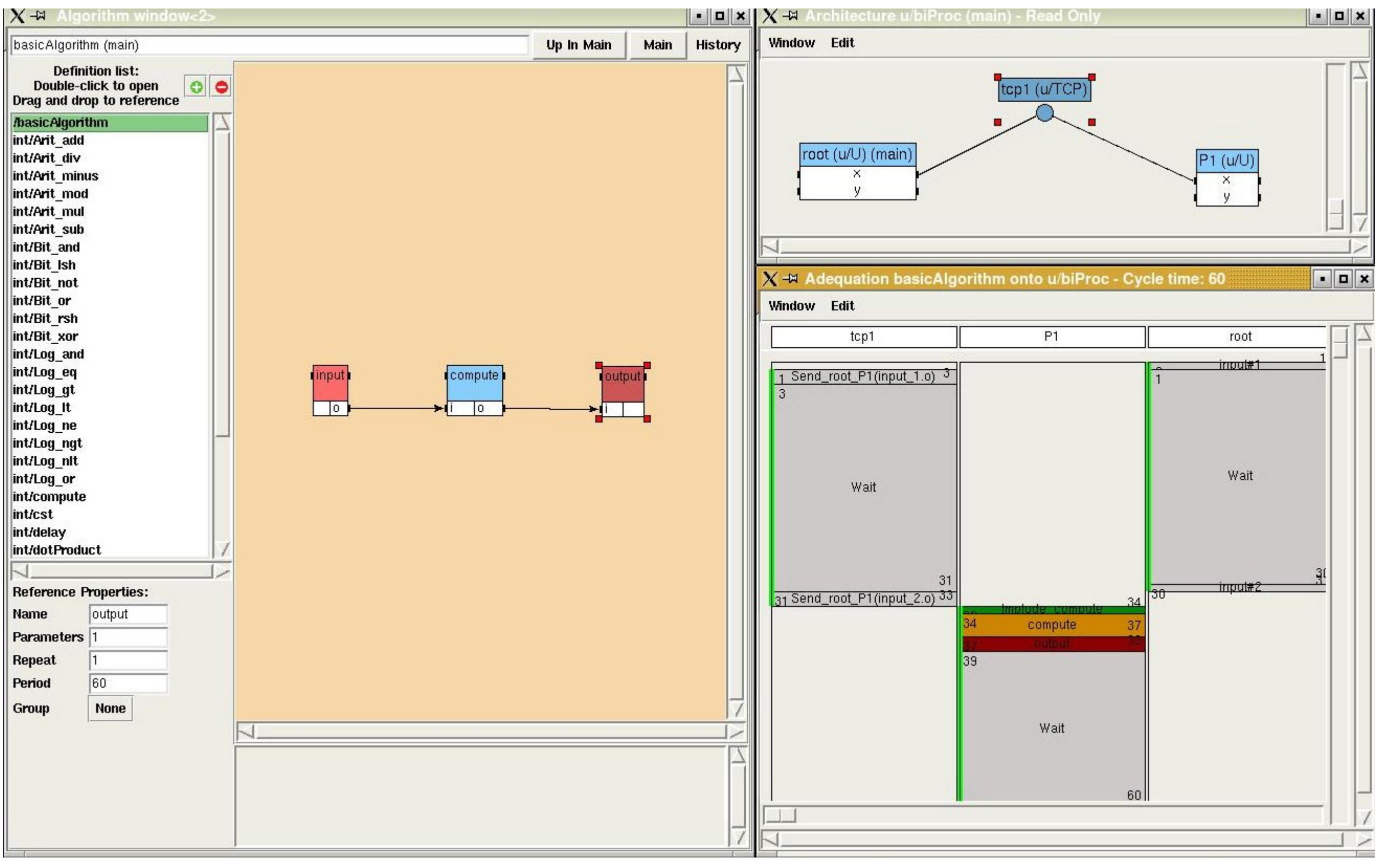
# SynDEx software

## GUI V6



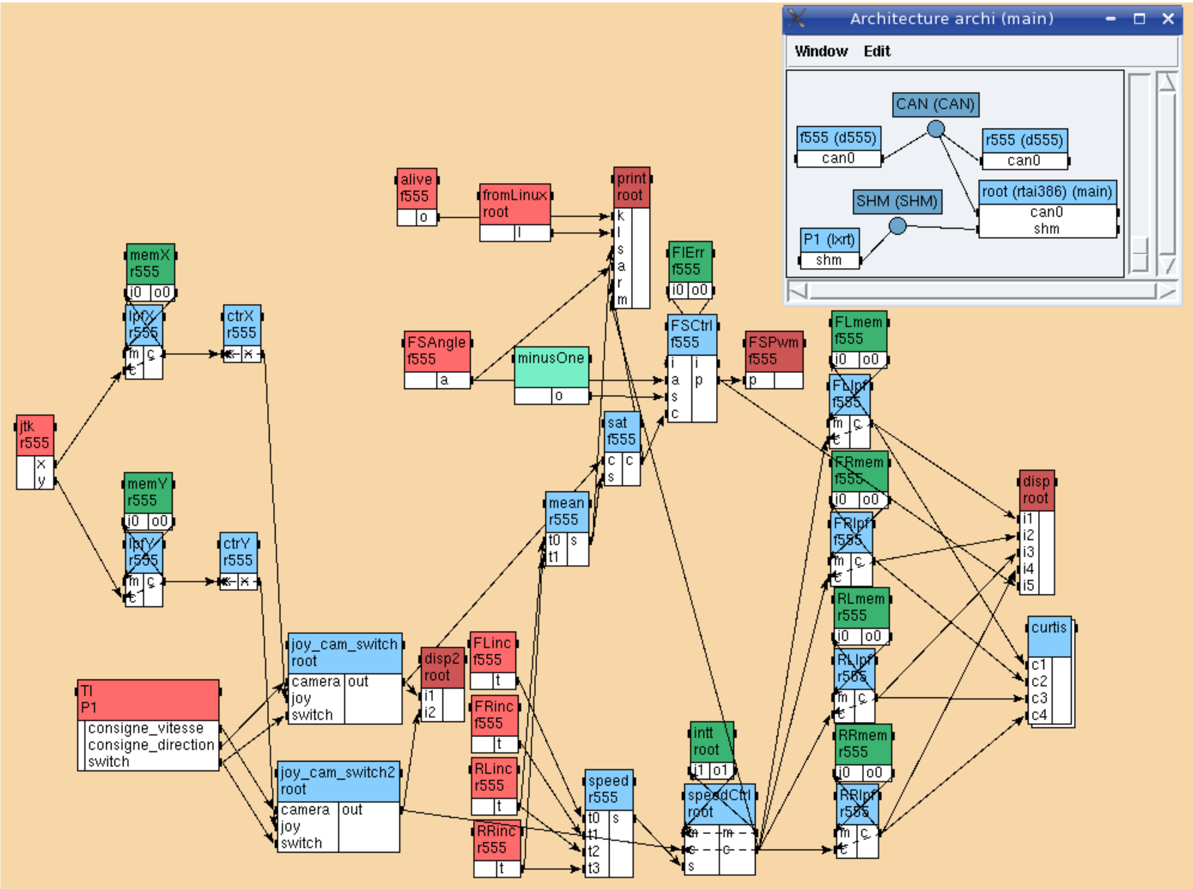

# SynDEx software

## GUI V7 (Simple algorithm: input 30ms, compute and output 60ms)



# SynDEx software

## GUI V7 (Automatic driving of CyCab: TI 100ms, ctrl-x 10ms)




# Conclusion

## Conclusion

- In order to perform an optimized implementation, we must master **links between control theory** and **computer science**.
- Embedded real-time systems must be **reactive**, **satisfy timing constraints** and **minimize resources**.
- The functional specification with some DSL allows **formal verifications**.
- The SIGNAL DSL verifies that the order of the output events is consistent with the order of the input events.
- The non functional specification allows the description of **hardware resources** and **timing characteristics**.
- The AAA methodology based on functional and non functional specifications, allows the formalization of implementations in terms of graph transformations, the study of implementations which are valid in terms of **schedulability**, the **minimization** of timing and resource criteria, and finally the **generation** of **safe by construction** embedded real-time executives.
- The SynDEx software **implements** the AAA methodology.

## Conclusion

**Safe by construction design**

**Optimized implementation: Adequation**

**Reduced development life cycle**