Systèmes temps réel: langages de programmation de systèmes temps réel

Claire Pagetti
claire.pagetti@enseeiht.fr

IN2 - 2014

ENSEEIHT - Département Télécommunication et Réseaux
2, rue Camichel, 31000 Toulouse

Objectifs du cours

- **Introduction au temps réel:**
  - –Qu'est-ce qu'un système temps réel?
- **Exemples de langages de programmation temps réel**
  - – SDL (System Description Language)
  - – Lustre – Scade
- **Introduction aux automates temporisés**

} C. Pagetti

- **Introduction aux RTOS**

} J. Ermont

- **Introduction à l'ordonnancement**

} J.-L. Scharbarg

Plan du cours

- **Cours 1: Introduction générale aux systèmes temps réel et aux problématiques de conception**
- **Cours 2 - 3: Langage Lustre**
- **Cours 4: RTOS**
- **Séances 5 - 7: TP RTOS / Lustre**
- **Cours 6: Langage SDL**
- **Cours 8 – 10: Automates temporisés**
- **Séance 11: TP sur UPPAAL**

Notation:

–Examen final le 25 mai 2014 – 16-18h

Plan

1. **Partie I - Introduction générale aux systèmes temps réel et aux problématiques de conception**
2. **Partie III – Langage Lustre: syntaxe, sémantique et preuve**
3. **Partie II - SDL**
4. **Partie IV – Introduction aux automates temporisés**

Partie I - Introduction générale aux systèmes temps réel

1. **Système temps réel**
   1. Définitions
   2. Exemple réel

2. **Architecture des systèmes temps réel**

3. **Calcul du WCET**

4. **Problématiques de conception des systèmes temps réel**

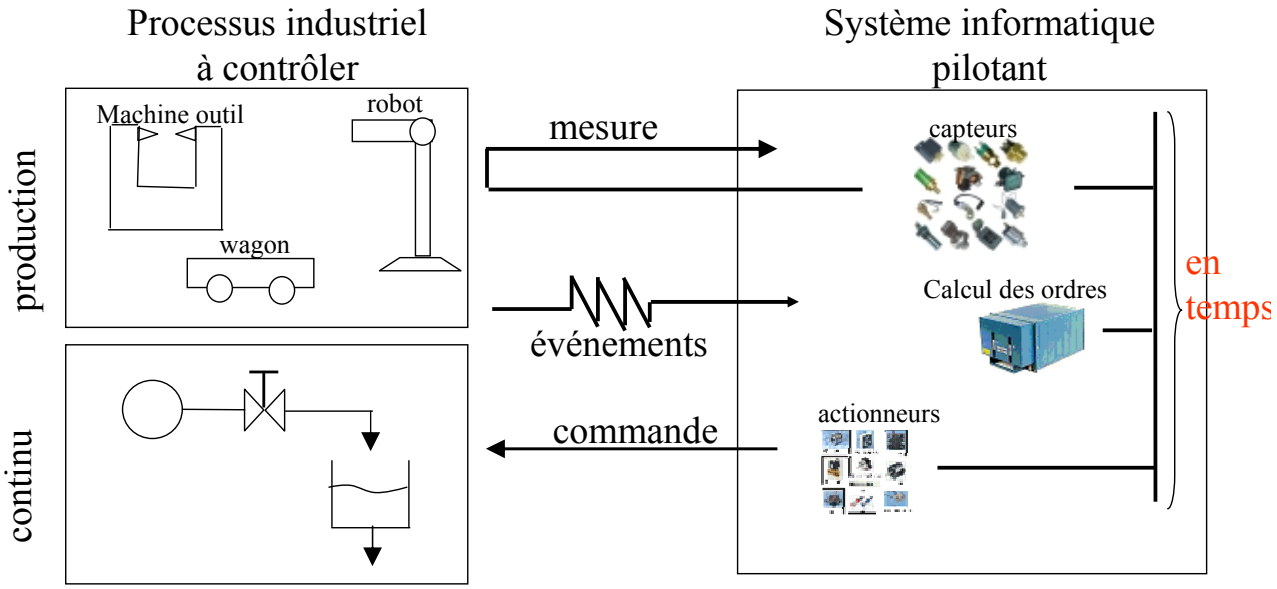
5. **Un peu de POSIX**

1.1 Introduction

- **La plupart des machines inventées par l'homme nécessitent un système de régulation ou de contrôle pour fonctionner**
- **Ces systèmes de contrôle existaient avant l'invention des ordinateurs**
- **Exemple:**
  - Pour maintenir une locomotive à vitesse constante, un système régule la quantité de vapeur. En descente, il faut injecter moins de vapeur et en montée, il faut en injecter davantage
- **Automatisation: réduire l'intervention humaine**

1.1 Première définition

Est appelé *temps réel* le comportement d'un système informatique dont le fonctionnement est assujetti à l'évolution dynamique d'un procédé industriel connecté à lui. On dit que le processus est contrôlé, piloté ou supervisé par le système qui *réagit* aux changements d'état du processus.



Historique – début années 60

•Le premier système moderne embarqué temps réel reconnaissable a été l'*Apollo Guidance Computer*, le système de guidage de la mission lunaire Apollo, développé par Charles Stark Draper du Massachusetts Institute of Technology.

•Premier ordinateur à avoir recours aux circuits intégrés

•Utilisé en temps réel par l'astronaute pilote pour recueillir et fournir des informations de vol, et pour le contrôle automatique de toutes les fonctions de navigation du vaisseau spatial.

•La mémoire utilise des mots de 16 bits : elle est composée de 64 ko (32 000 mots) de mémoire morte contenant l'ensemble des programmes et de 4 ko (2 000 mots) de mémoire vive (effaçable) utilisée par les traitements. Les deux types de mémoire sont constituées de tores magnétiques : les programmes sont implantés dans l'ordinateur à la fabrication. Le processeur lui est constitué de plus de 5 000 portes NOR réalisé à l'aide de circuits intégrés. Il pèse environ 35 kg.

Source: http://fr.wikipedia.org/wiki/Apollo_Guidance_Computer

Embedded system market

"Over 4 billion embedded processors were sold last year and the global market is worth €60 billion with annual growth rates of 14%. Forecasts predict more than 16 billion embedded devices by 2010 and over 40 billion by 2020.

Embedded computing and electronics add substantial value to products. Within the next five years, the share of embedded systems are expected to increase substantially in markets such as automotive (36%), industrial automation (22%), telecommunications (37%), consumer electronics (41%) and health/medical equipment (33%). The value added to the final product by embedded software is much higher than the cost of the embedded device itself. For example, in the case of a modern car, by 2010 over 35% of its value will be due to embedded electronics. "

[ARTEMIS JU Organisation - 2010]

Temps réel ≠ aller vite

–besoin d'un temps de réaction court (1 ms) pour le contrôle d'un avion de combat

–besoin d'un temps de réaction moins court (10 ms) pour le contrôle d'un avion de transport civil

–besoin d'un temps de réaction moins court (1s) pour une IHM

–besoin d'un temps de réaction moins court (1mn) pour le contrôle d'un chaîne de production (lente)

–besoin d'un temps de réaction moins court (1h) pour le contrôle d'une réaction chimique (lente)

–…

–besoin d'un temps de réaction de quelques heures pour l'établissement d'une prévision météorologique

–besoin d'un temps de réaction de quelques jours pour le calcul de la paie…

=> l'important est de respecter l'échéance

=> un résultat juste hors délai peut être inutilisable et considéré comme une faute.

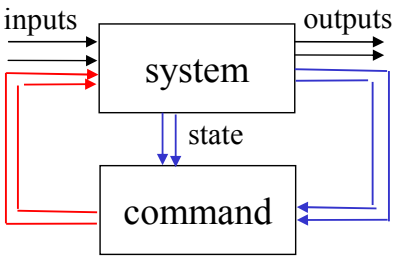
Control-command

Command: Laws which govern the dynamical evolution of a system

- Command of actuators regarding the sensors
- In continuous time

Examples:

- Regulation of a liquid level between thresholds
- Command of flight surfaces



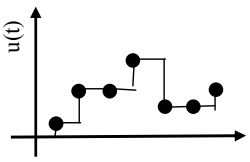
Equations of state

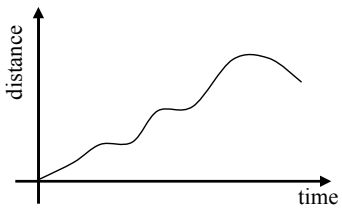
$$\begin{cases} dx = f(x,u) \\ y = h(x) \end{cases}$$

x internal state, y output, u input

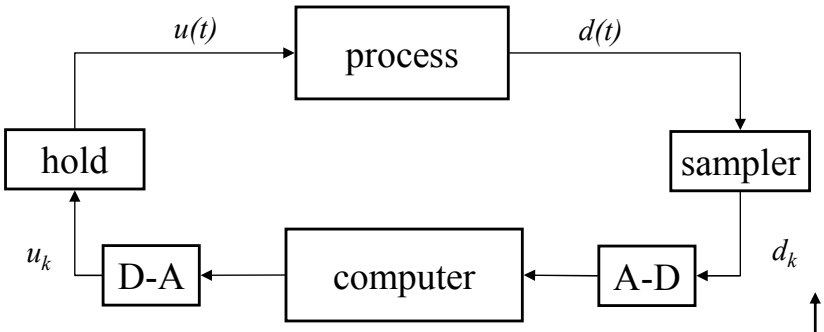
Command a system = make the system evolve in order to reach a particular configuration or to follow a given trajectory

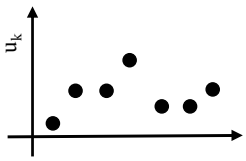
Boucle d'asservissement





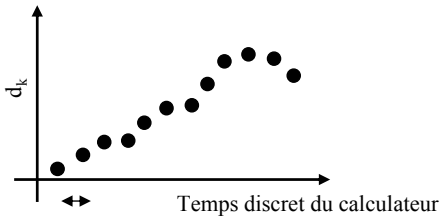
vitesse: $v = \frac{d(\text{distance})}{dt}$





Ordre élévateur

$$v_k = \frac{d_k - d_{k-1}}{T}$$



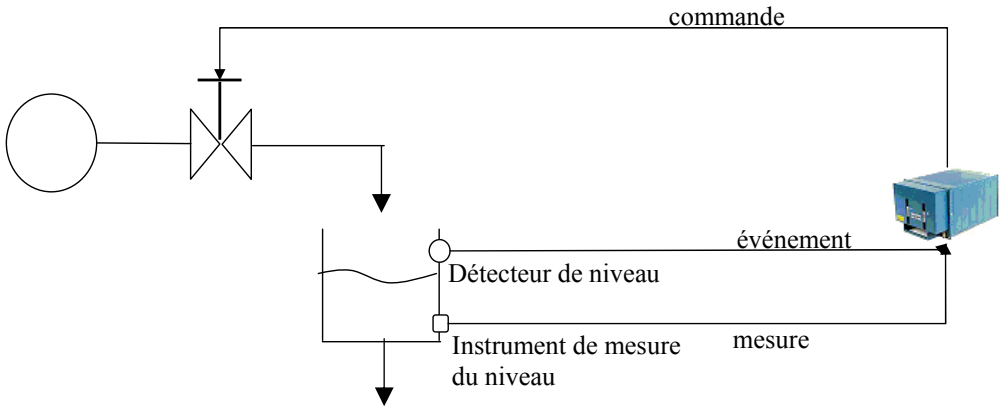
T période d'échantillonnage

Exemple: commande d'un procédé continu

Régulation du niveau d'un liquide:

•Contrôle du niveau de façon à respecter un ordre de l'opérateur

•Respecter le niveau toléré



Temps réel dur et mou

Deux notions de criticité face au manquement d'une échéance

–Contraintes temps réel **strictes** : le dépassement d'une échéance est catastrophique

–Contraintes temps réel **relatives** : le dépassement d'une échéance peut être toléré (dans une certaine mesure)

⇒**Temps réel dur** / **temps réel lâche** !

Conséquences…

=> dans le cas des systèmes temps réel dur, on cherchera à être **prévisible**, **déterministe** et **fiable**

=> utilisation de techniques mathématiques (ordonnancement, évaluation des pires cas…)

=> dans le cas des systèmes temps réel lâche, on cherchera à minimiser la probabilité de rater une échéance plusieurs fois de suite...

Définitions

Prédictibilité (predictability):

–Les performances de l'application doivent être définies dans tous les cas possibles de façon à assurer le respect des contraintes de temps. On parle de pire cas.

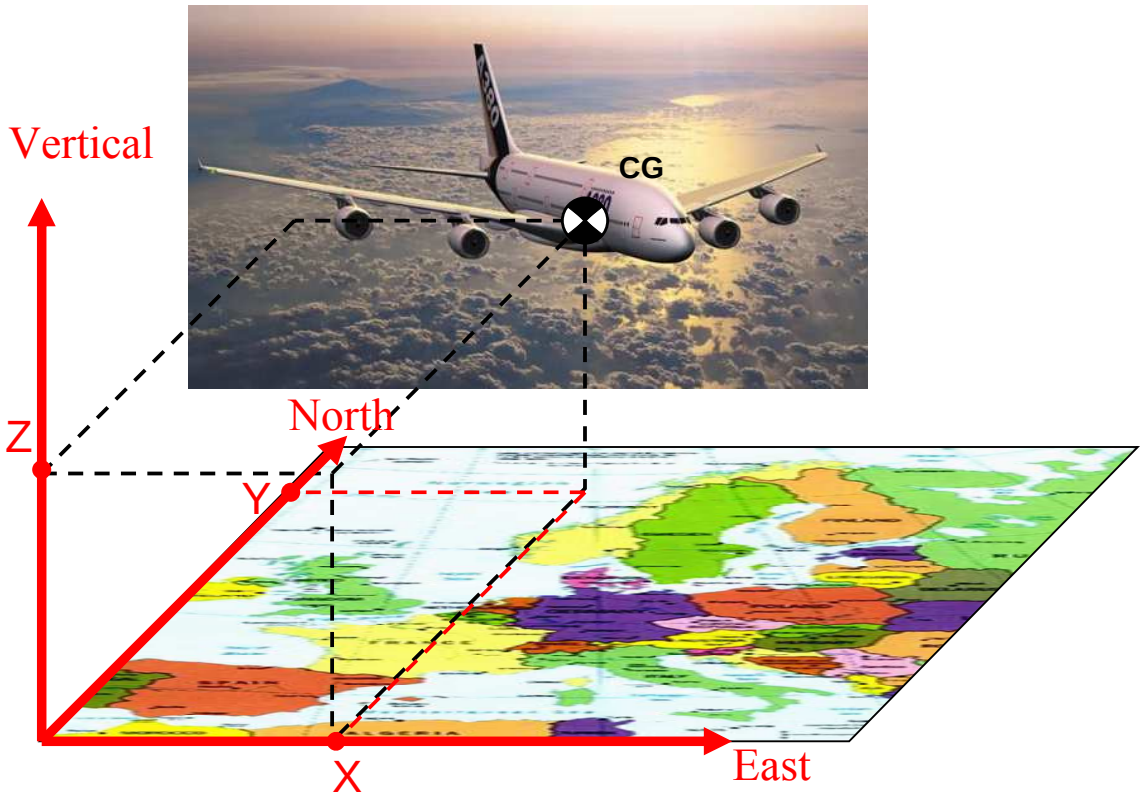
Déterminisme (determinism):

–Il n'y a aucune incertitude sur le comportement du système: pour un contexte donné le comportement est toujours le même.

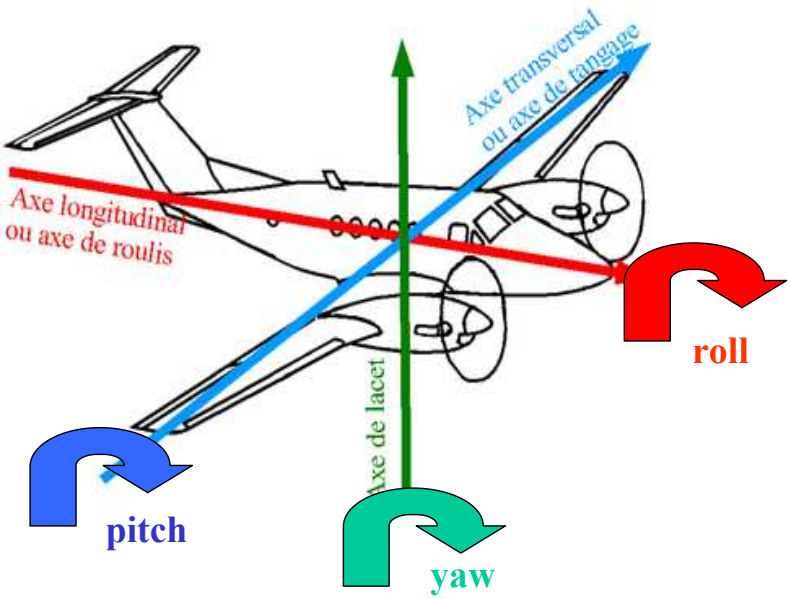
Fiabilité (reliability):

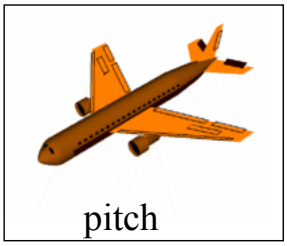
–Capacité d'un système à réaliser et maintenir ses fonctionnalités dans des conditions normales d'utilisation. En temps réel, la fiabilité concerne le respect des contraintes temps réel. On peut également vouloir que le système reste fiable même si certaines pannes sont apparues, on parle alors de tolérance aux fautes.

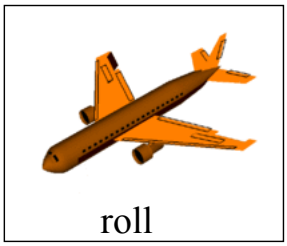
1.2 Exemple d'un système temps réel dur: les commandes de vol d'un aéronef

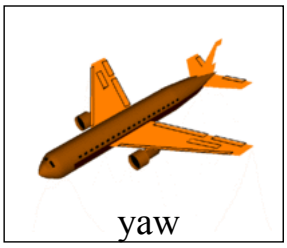


Contrôle de l'aéronef







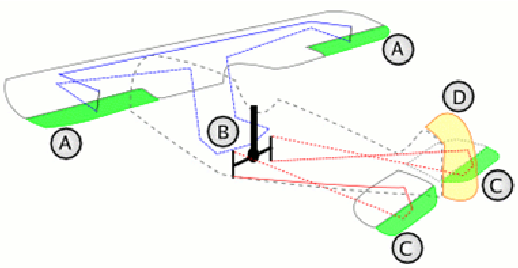


Flight control system

The **aircraft primary flight control system** is the set of elements between the stick and the surfaces which aim at controlling the attitude, the trajectory and the speed of the aircraft.

The system is composed of:

–piloting elements: stick (or yoke or control column), pedals, throttle controls…



–transmitting and computing organs

- cables and calculators (for fly-by-wire)

–sensors, actuators and servocommand to command the surfaces

Système de commande de vol (CDV)

(Some) Aircraft sensors

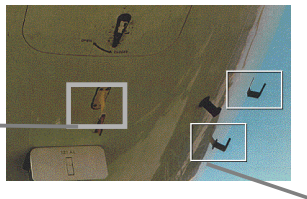
–GPS

–altimeter: measure the altitude of an object above a fixed level

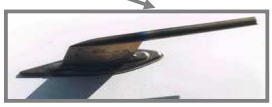
–inertial measurement unit is an electronic device that measures and reports on a craft's velocity, orientation, and gravitational forces, using a combination of accelerometers and gyroscopes.

An attitude indicator, also known as gyro horizon or artificial horizon, is an instrument used in an aircraft to inform the pilot of the orientation of the aircraft relative to earth. It indicates pitch and roll.

A weather vane, also known as a peach patrolwind vane or weathercock, is an instrument for showing the direction of the wind.



A pitot tube is a pressure measurement instrument used to measure fluid flow velocity



A320 flight controls systems

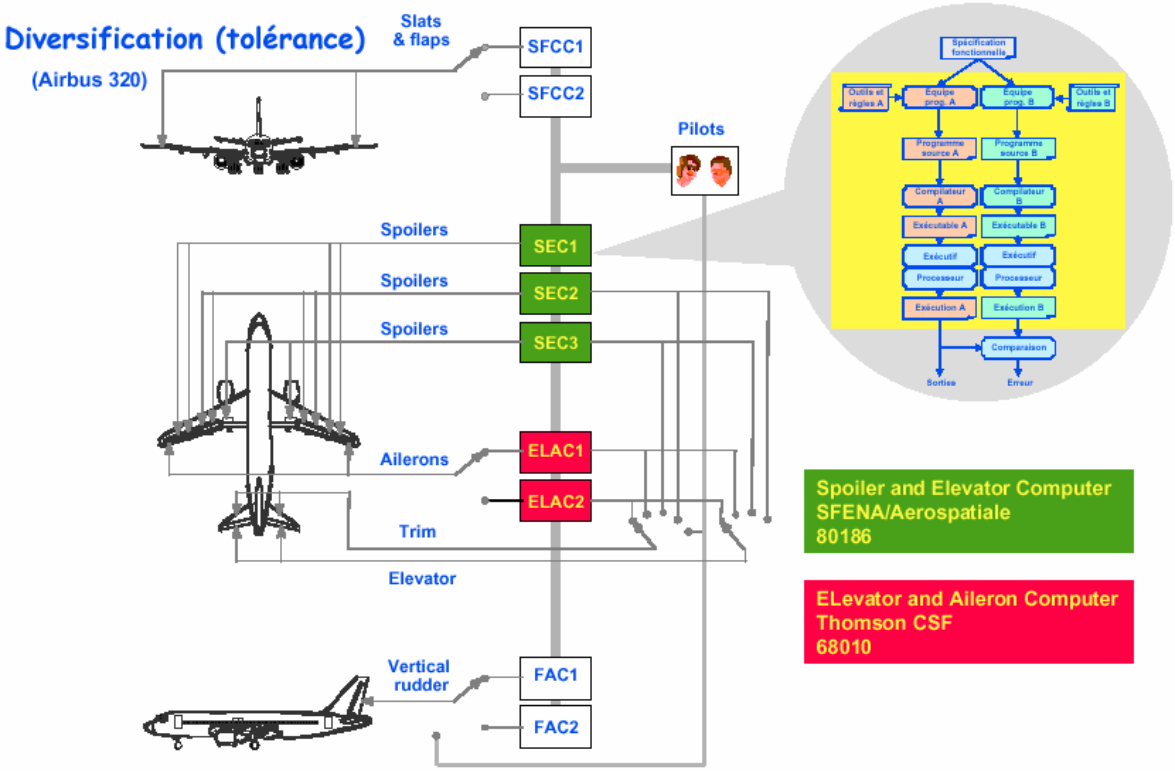
Fly-by-wire system

–9 calculators

- Functions allocation :
  - –2 redundant calculators for the slats and the flaps (SFCC1-2)
  - –2 redundant calculators for the rudder (FAC1-2)
  - –3 redundant calculators the spoilers, the elevators and the trim (SEC1-2-3)
  - –2 redundant calculators for the ailerons, the elevators and the trim (ELAC1-2), replaced in case of failure by the SEC1-2-3

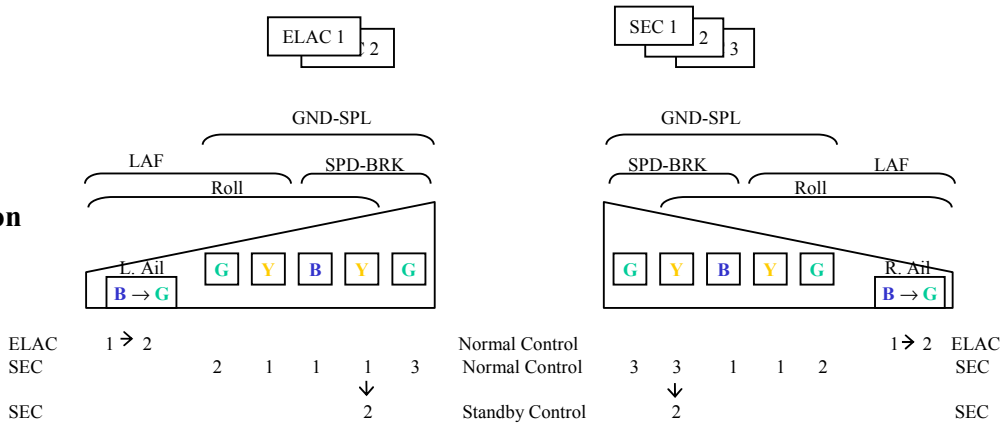
- Safety requirements
  - –each calculator must be "fail-silent"
  - –each calculator must have a failure rate less than 10^{-3} per flight hour
  - –...

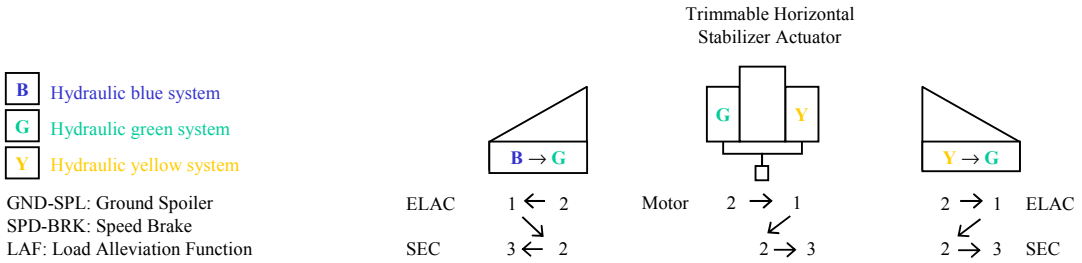
A320 flight controls systems architecture



Reconfiguration policies

Redundancies allocation

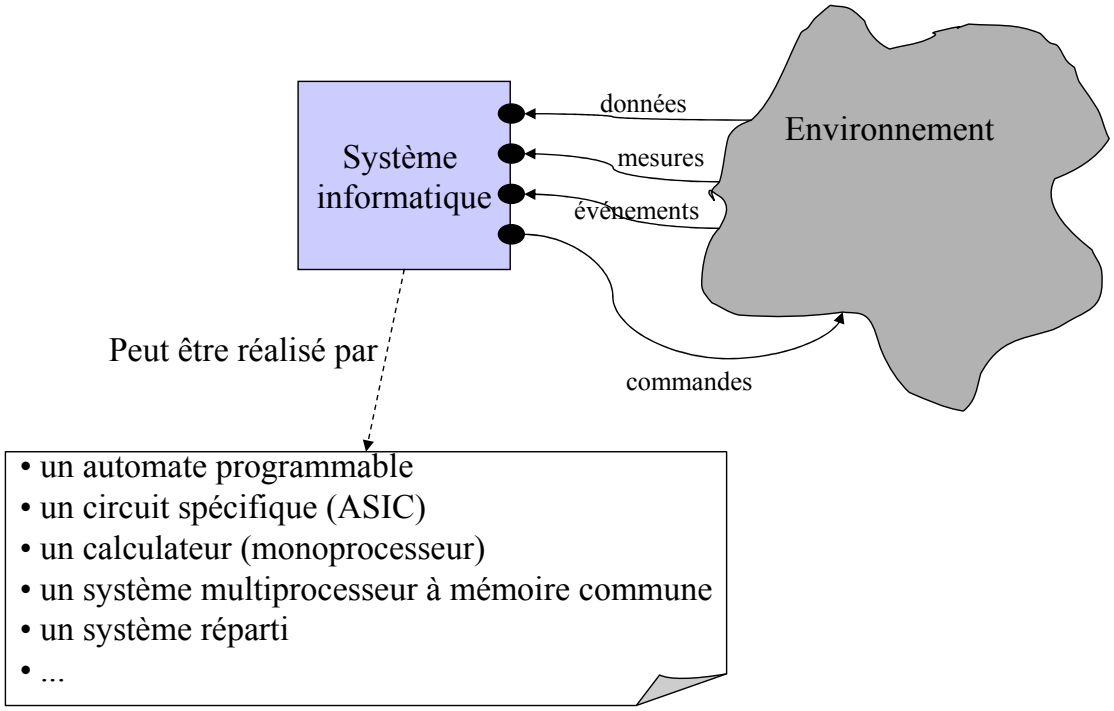




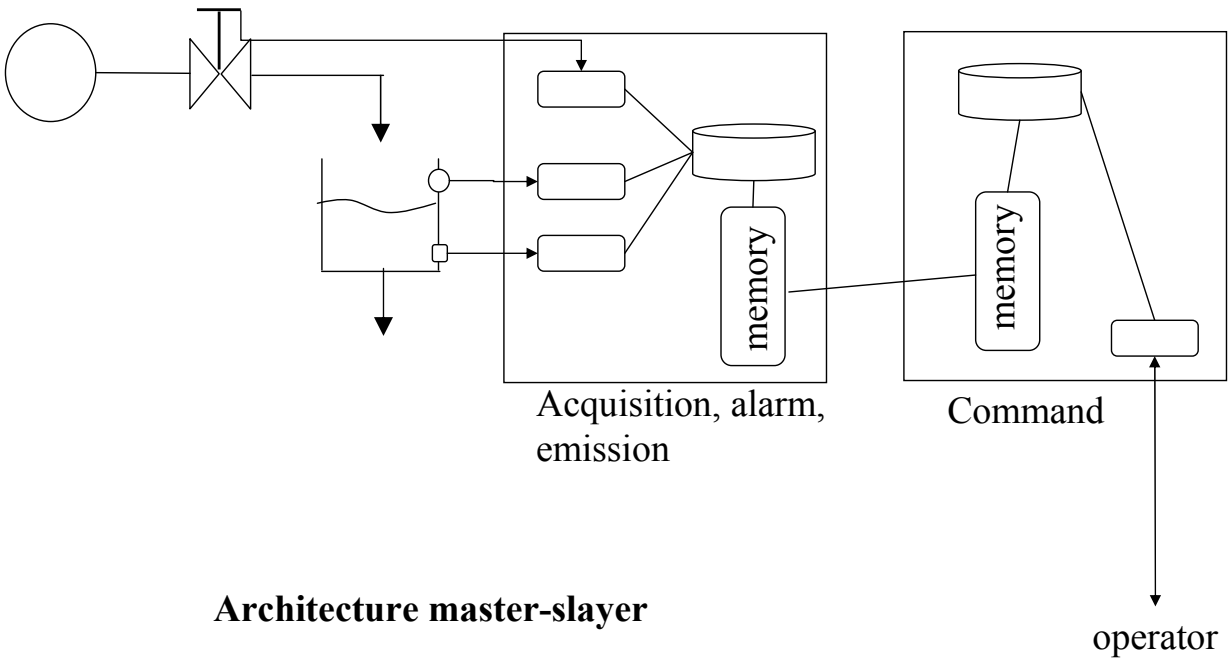
Plan

1. **Systèmes temps réel**
2. **Architecture des systèmes temps réel**
   1. Architecture matérielle
   2. Comportement fonctionnel
   3. Support exécutif
3. **Calcul de WCET**
4. **Problématiques de conception de systèmes temps réel**
5. **Un peu de POSIX**

2.1 Architectures matérielles

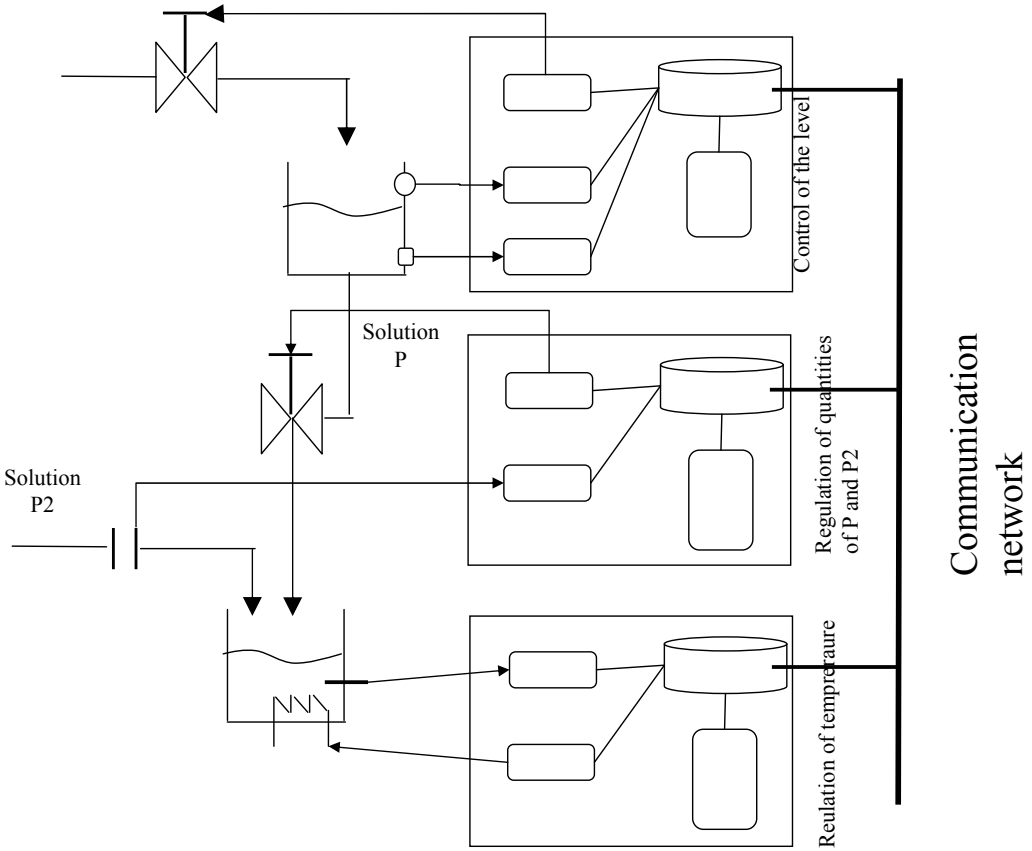


Architecture multiprocesseur



Architecture master-slayer

Architecture distribuée



2.2 Programmer des systèmes temps réel

- **Difficultés classiques de programmation**
- **… auxquelles s'ajoutent les difficultés de gestion du temps**
  - Datation des signaux
  - Ordre d'exécution des actions
  - Attente d'un signal pendant un délai …

  Et: le monde est plein de délais, d'échéances, de dérives d'horloge …

Exemple typique:

Dans un contexte où les actions prennent un certain temps, comment doit être interprétée une absence d'information?

- L'absence peut être due à:
  - un délai
  - une absence réelle
- Une incompréhension de la situation
- …

2.2 Architecture fonctionnelle

Fonctionnement général : boucle infinie

```
Tant que TOUJOURS faire
  Acquisition des entrées (données capteurs, mesures…)
  Calcul des ordres à envoyer au procédé
  Émission des ordres
Fin tant que
```

Mais, deux modes de fonctionnement :

–fonctionnement **cyclique** (time driven ou système "synchrone" (mais pas le même "synchrone" que les langages "synchrones"))

–fonctionnement **événementiel** (event driven)

=> fonctionnement mixte : à base de traitements périodiques et et apériodiques (déclenchés sur événements)

Fonctionnement cyclique

Scrutation périodique d'une mémoire d'entrée (polling)

⇒échantillonnage des entrées sur l'horloge du système

⇒activation du système à chaque top d'horloge

```
A chaque top d'horloge faire
    Lecture de la mémoire des entrées
    Calcul des ordres à envoyer au procédé
    Émission des ordres
Fin
```

Mais :

–système peu "réactif" si l'environnement produit des informations à des fréquences différentes

=> oblige à prévoir toutes les réactions du système dans la même boucle

=> problème de performance

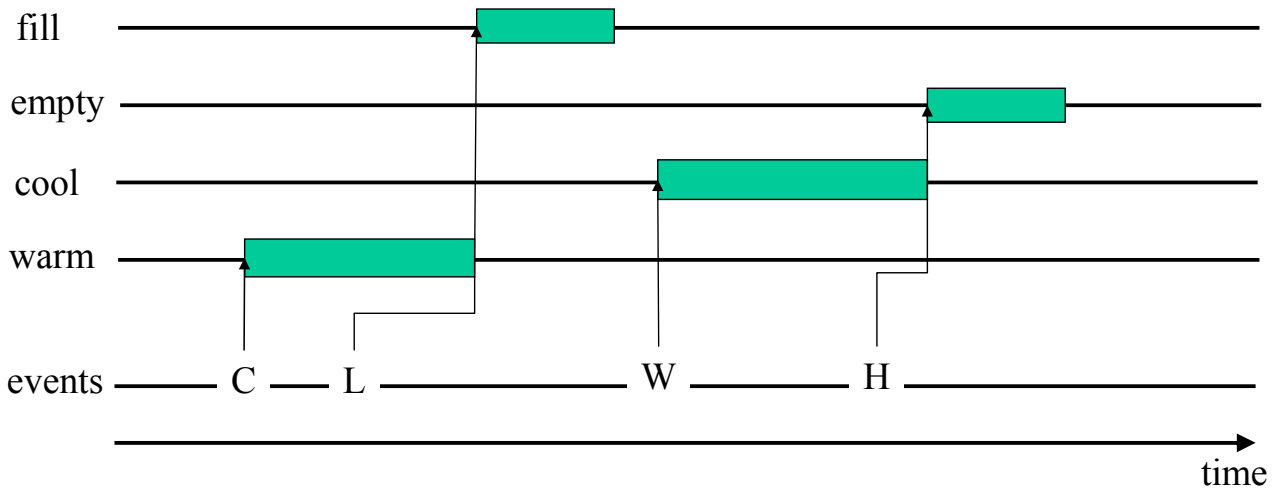
=> ou oblige à imbriquer des boucles de fréquences multiples

=> difficultés de réalisation, de lisibilité du code, d'évolution

Chronogramme d'une implémentation cyclique

Régulation du niveau et de la température d'un liquide

•Evénements: H (for high), L (for low), C (for cold) and W (warm)



Fonctionnement événementiel

Activation du système à chaque événement (=> notion d'interruption)

```
A chaque interruption faire
    Lecture de l'information arrivée
    activation du traitement correspondant
    Émission des ordres issus de ce traitement
Fin
```

Mais : **que faire si une interruption survient alors que le système est en train de traiter une interruption précédente ?**

=> notion de priorité des interruptions

=> notion de "tâche" associée à une ou plusieurs interruptions

=> mécanisme de préemption et de reprise de tâche

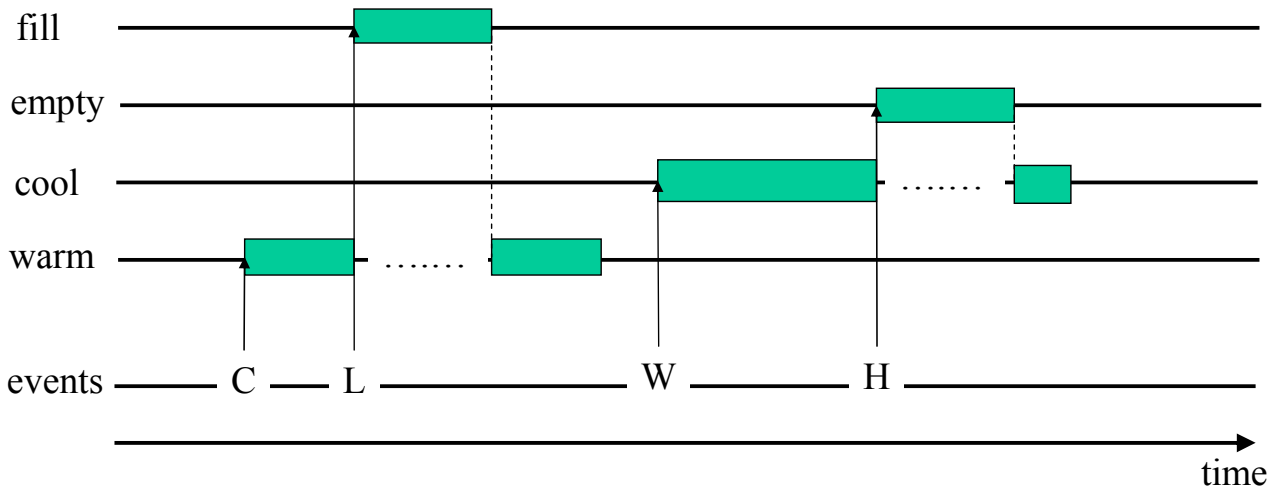
=> gestion de l'exécution concurrente des tâches (ordonnancement)

=> Un système temps réel est souvent un système multitâche incluant un gestionnaire de tâches (Ordonnanceur)

Chronogramme d'une implémentation événementielle

Régulation du niveau et de la température d'un liquide

- **Events: H (for high), L (for low), C (for cold) and W (warm)**
- **Priorités: 2, 2, 1, 1**



Ressources

Ressource:

–Entité utilisée par un système informatisé pour un bon fonctionnement

- Physique: capteurs, actionneurs, périphériques …
- Logique: donnée dans une mémoire, code …

Ressource partagée:

–Ressource potentiellement partagées par plusieurs tâches qui peuvent d'exécuter en parallèle. Besoin de cohérence de la ressource.

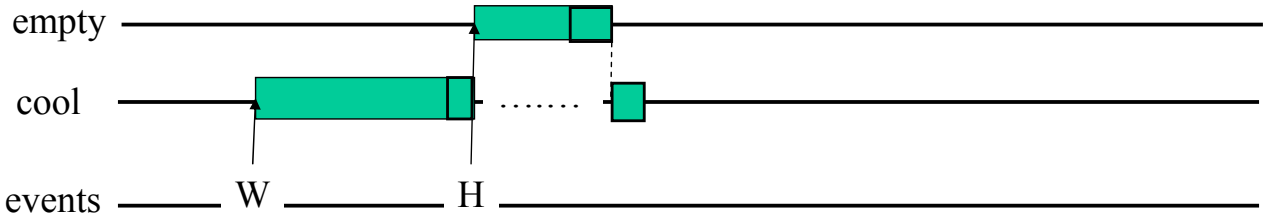
Ressource critique:

–Accès par au plus une tâche à la fois. Besoin de mécanisme d'exclusion mutuelle de la ressource (sémaphore, …)

Section critique:

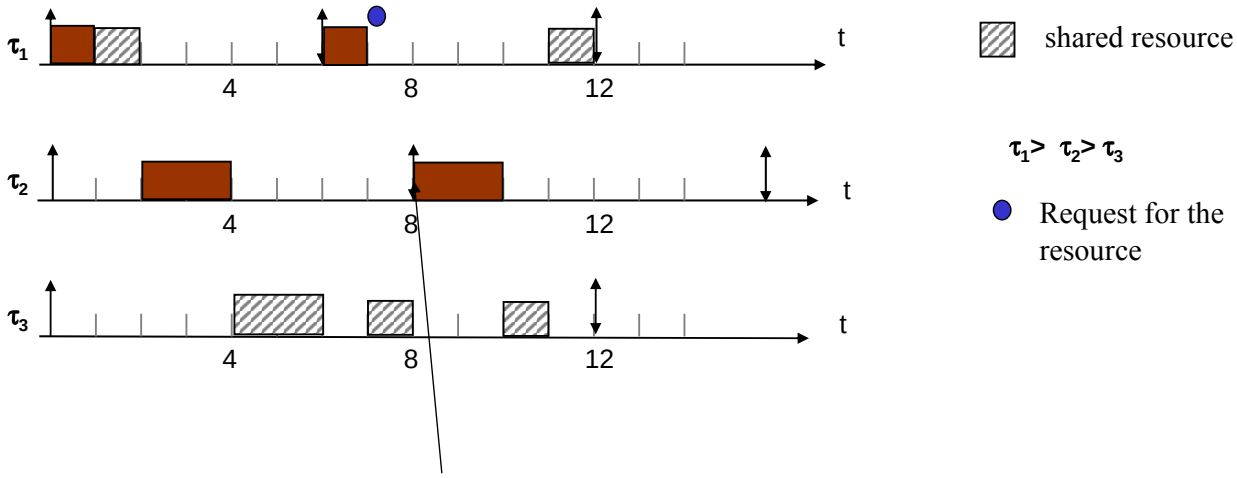
–Zone du code où une tâche fait un accès à une ressource partagée critique.

Ressource partagée



Quand elle est interrompue, la tâche cool était en train d'accéder à une ressource partagée (par exemple une mémoire). Il y a donc un conflit.
=> Une solution: interdiction de préempter une tâche quand elle utilise une ressource partagée critique.

Priority inversion



Priority inversion

Example: Sojourner

The Sojourner rover was the second space exploration rover to successfully reach another planet, and the first to actually be deployed on another planet. Sojourner landed on Mars as part of the Mars Pathfinder mission on July 4, 1997.

[http://en.wikipedia.org/wiki/Sojourner_%28rover%29]

Priority inversion:

"Even though NASA knew the problem, because it already occurred on all the test that had been performed. But NASA thought it won't be a problem because on earth the situation didn't occur very often, NASA only underestimated the number of situations the problem appeared."

Robert Franz. Advanced Course Seminar Analysis of computer bugs, Priority Inversion Mars Sojourner. 2008.

Exemple: algorithme de Peterson

Processus P0

```
flag[0] = 1
turn = 1
while(flag[1] && turn == 1);
// do nothing
// critical section
   ...
flag[0] = 0
```

Processus P1

```
flag[1] = 1
turn = 0
while(flag[0] && turn == 0);
// do nothing
// critical section
   ...
flag[1] = 0
```

L'algorithme de Peterson est un algorithme d'exclusion mutuelle à base d'attente active permettant à deux processus de partager une ressource sans conflit.

L'algorithme utilise deux variables, *flag* et *turn*. flag = 1 indique que le processus veut accéder à la section critique. La variable *turn* vaut l'identité du processus qui l'utilise. L'entrée dans la section critique est assurée pour le processus P0 si P1 ne veut pas entrer dans la section critique ou si P1 a donné la priorité à P0 en mettant *turn* à 0.

Exemple: Drone Paparazzi

Pararazzi est un projet opensource pour programmer des drones (ou UAV pour Unmanned Air Vehicle System). Développé à l'ENAC.

http://paparazzi.enac.fr/wiki/Main_Page

L'environnement fournit le code pour:

-la maquette

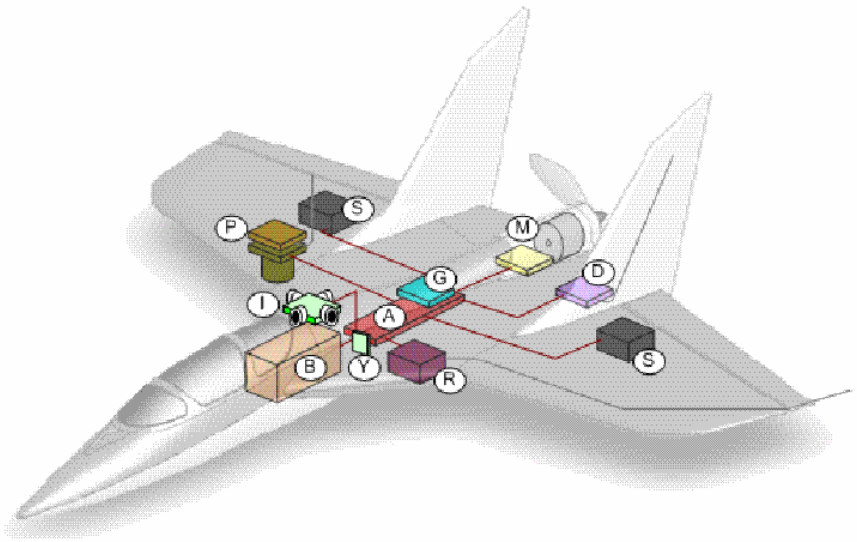
-maquette du commerce de 250 g à 1.4 kg et de 0.3m à 1.4 m.

-le simulateur de vol

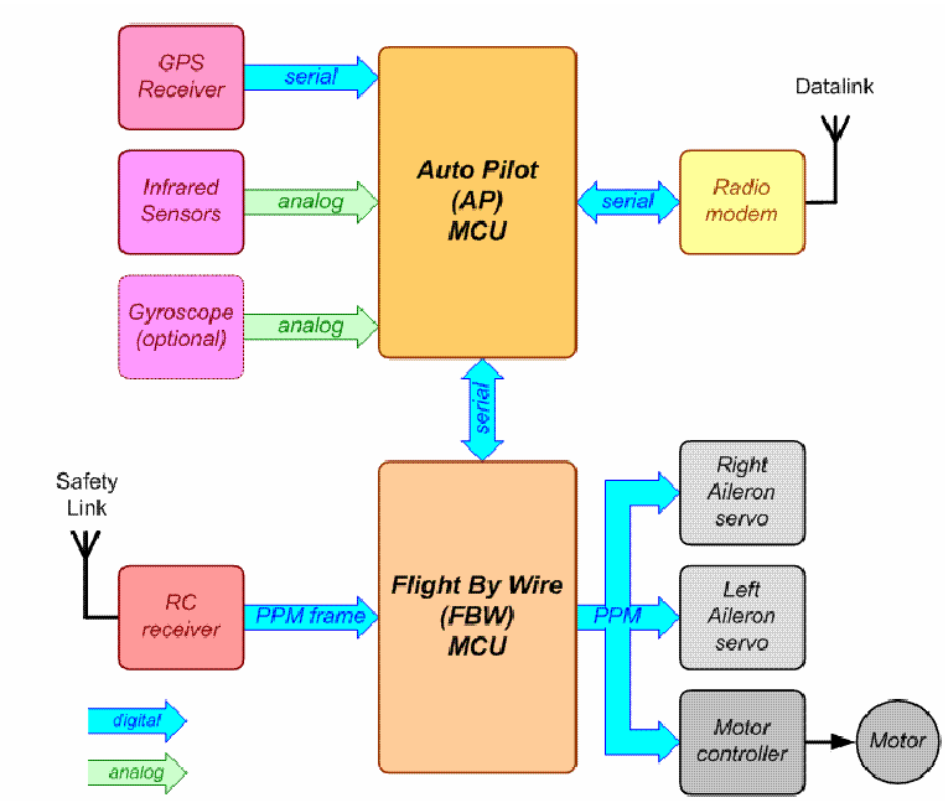
Exemple: drone Paparazzi

Architecture hardware

A **utopilot**
B attery
D **atalink**
G **PS**
I **nfrared**
M otor
P **ayload**
R eceiver
S ervo
GY **ro**



Architecture fonctionnelle



Code de la boucle d'exécution

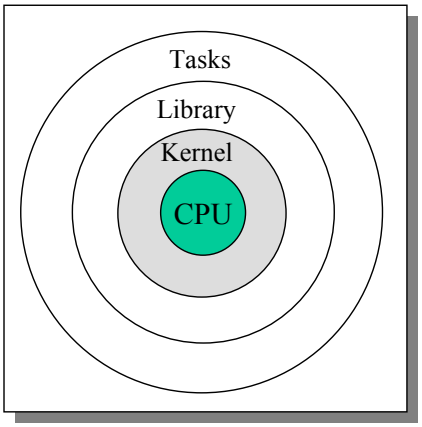
No Operation System

Main Loop

```
int main( void ) {
  Fbw(init);
  Ap(init);
  InitSysTimePeriodic()
  while (1) {
    if (sys_time_periodic()) {
      Fbw(periodic_task);
      Ap(periodic_task);
    }
    Fbw(event_task);
    Ap(event_task);
  }
  return 0;
}
```

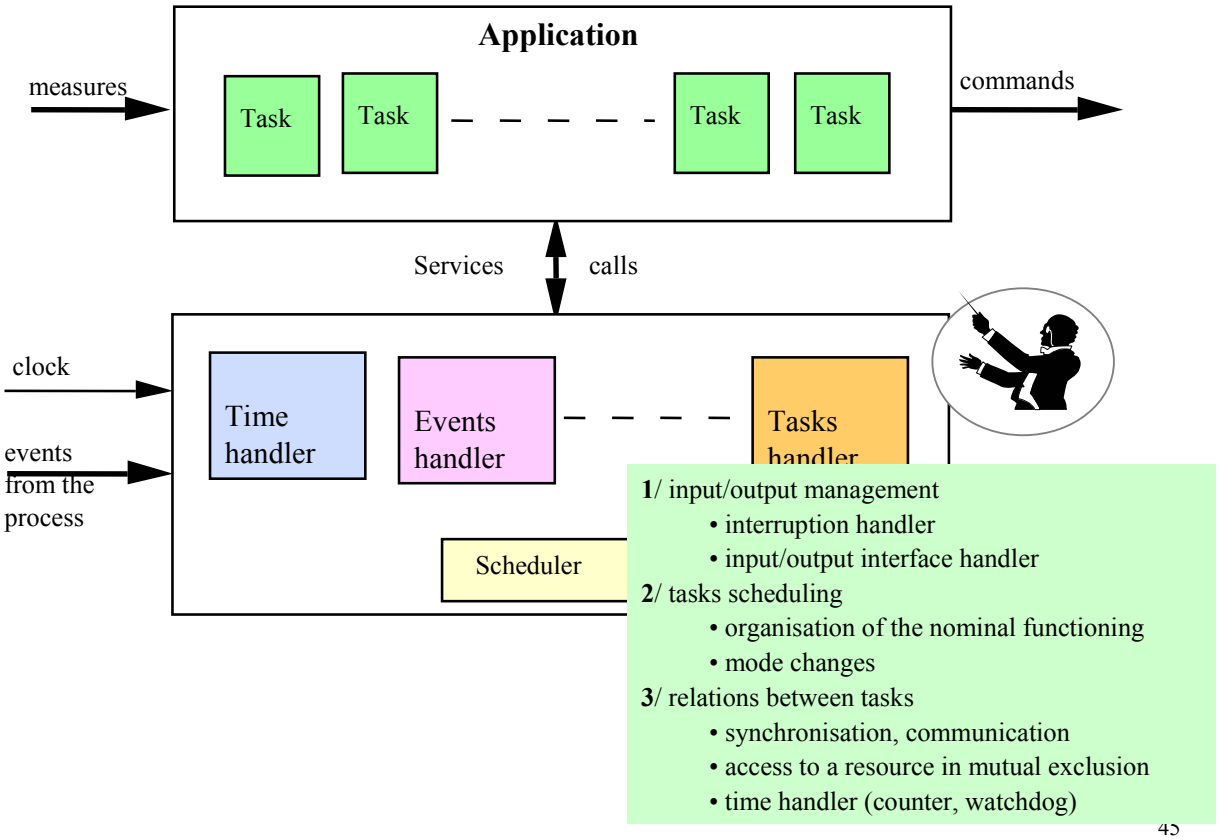

2.3 Support exécutif temps réel

Solution la plus simple: application nue sur le calculateur



Polling des données
Librairies de fonctions déterministes
Séquencement à la main des tâches

Solution avec un exécutif temps réel



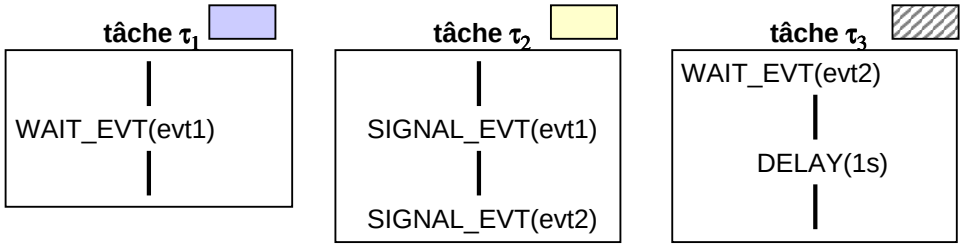
OS temps réel

- **OS pour le temps réel dur**
  - –Gestion précise des priorités
  - –Exécution en temps borné des primitives système (interruptions, sémaphores…)
  - –Pas de mémoire virtuelle
  - –Minimisation des « overhead » (temps pris par le système pour exécuter et surveiller son propre comportement)

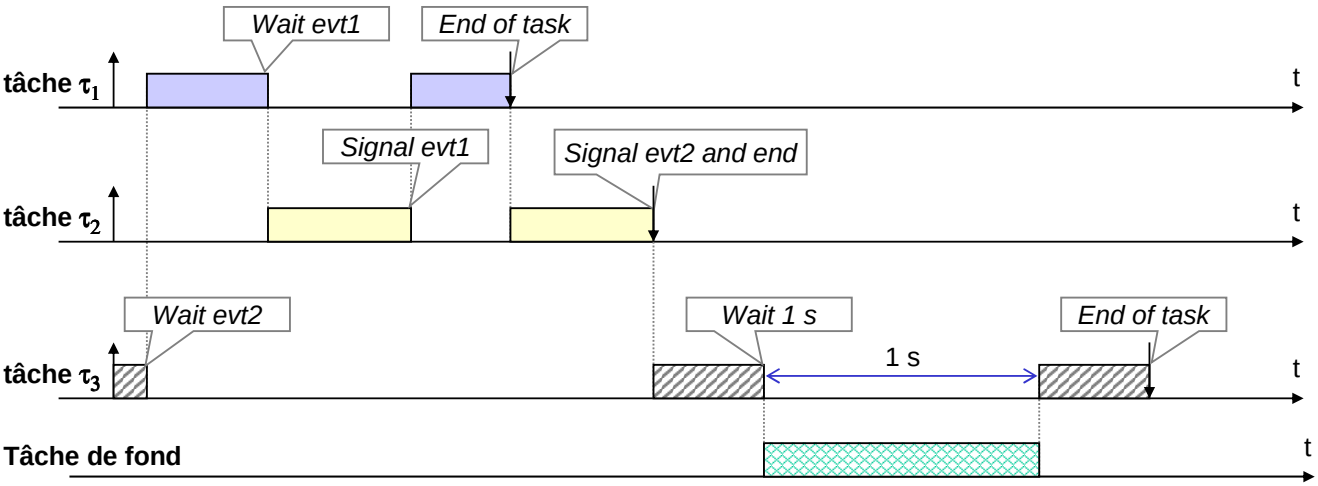
- **OS pour le temps réel mou**
  - –Extension des OS classiques (comme UNIX..)
  - –Extension des ordonnancements temps réel
  - –Processus légers
  - –Préemption …

Coopération entre tâche et ordonnancement : exemple

Configuration : (3 tâches)



Ordonnancement Prio(τ_1)=2 - Prio(τ_2)=1 - Prio(τ_3)=3



Coopération entre tâches

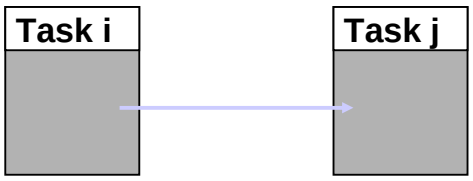
Synchronisation:

–Evénements (attente d'un événement, envoi d'un événement)

–Sémaphore (Demander / Libérer un sémaphore)

–Rendez-vous

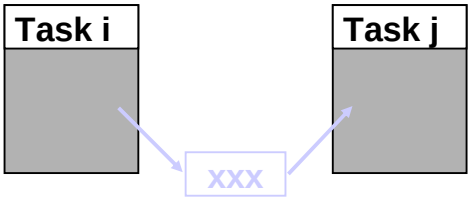
–…



Communication

–Boîte à lettres (écrire / lire une donnée)

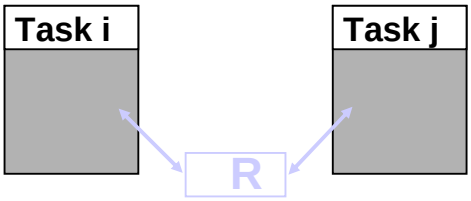
–…



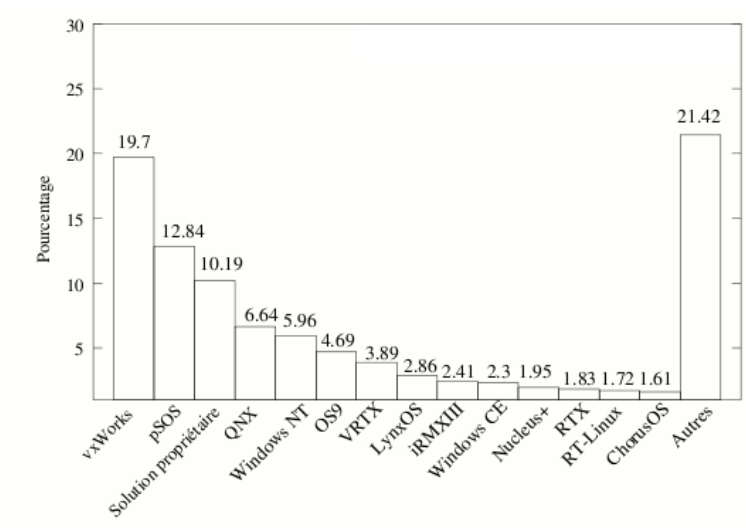
Partage de ressources critiques

–Sémaphore en exclusion mutuelle

–…



Marché pour les OS temps réel



[TIM00] M. Timmerman. « RTOS Market survey : preliminary result ». Dedicated System Magazine, (1):6–8, January 2000.

Exemple: APEX Arinc 653

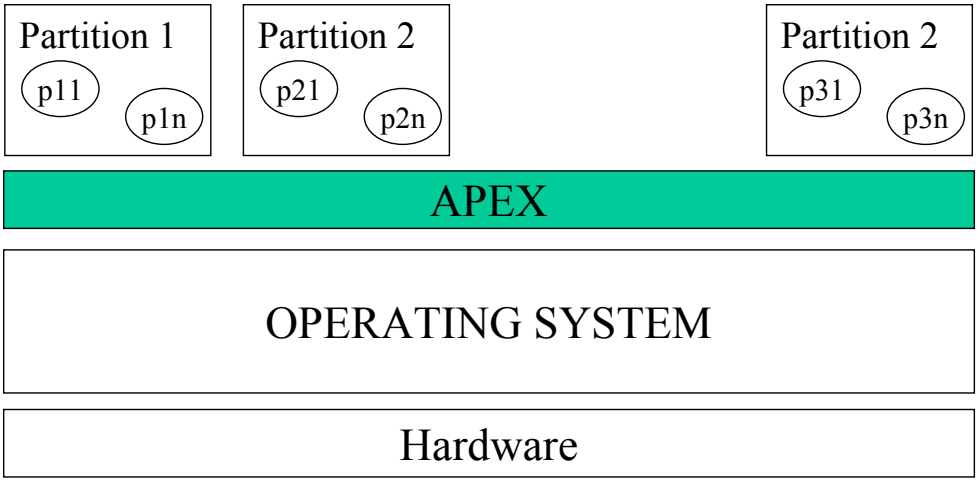
- **Arinc (Aeronautical Radio, Incorporated) créé en 1929, est une société détenue par les principales compagnies aériennes et des constructeurs aéronautiques américains qui est connue pour détenir les principaux standards de communications à l'intérieur des aéronefs et entre les aéronefs et le sol.**

- **Publication du premier standard ARINC 653 : été 1996**

- **Premiers travaux : 1991**

- **Arinc 653: Application programming interface (API) entre un OS d'une ressource avionique et une application. L'implémentation Airbus s'appelle l'APEX (APplication / EXecutive)**

Fonctionnement de l'A653



Operating system :

–Schedule the module partitions
–Schedule the processes of each partition
–Ensure segregation (partitioning) spatially and temporally

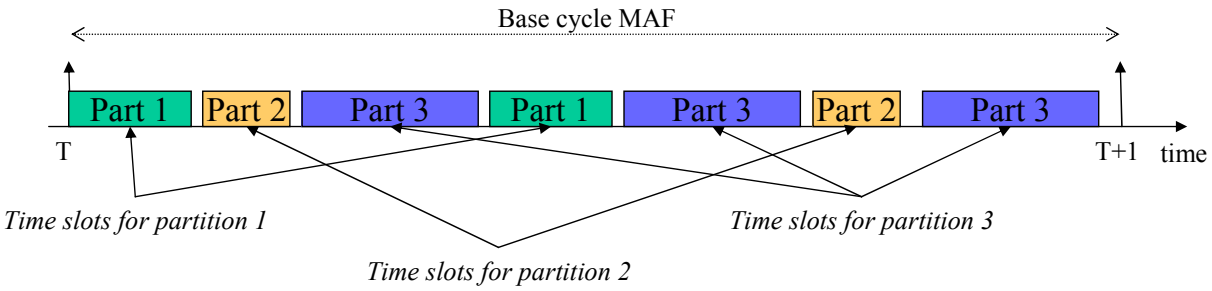
Partage des ressources

Ségrégation spatiale:

Zone mémoire allouée statiquement pour chaque partition.

Ségrégation temporelle (CPU) :

–Une partition n'a pas de priorité
–Attribution déterministe et cyclique du CPU à chaque partition:
=> OS répète le cycle de base de durée fixe (MAjor time Frame: MAF)
=> Allocation d'un slot de temps de la MAF pour chaque partition



⇒L'allocation des ressources se fait statiquement hors ligne et ne peut être modifiée.

Partie I - Introduction générale aux systèmes temps réel

1. **Système temps réel**

2. **Architecture des systèmes temps réel**

3. **Calcul du WCET**
   1. Méthode d'analyse statique
   2. Implantation prédictible sur multi-cœurs
   3. Et la suite?

4. **Problématiques de conception des systèmes temps réel**

5. **Un peu de POSIX**

The WCET Problem

Given

- ❑ the code for a software task
- ❑ the platform (OS + hardware) that it will run on

Determine the WCET of the task.

Can the WCET always be found?

- ❑ In general, no, because the problem is undecidable.

Usual restrictions in embedded systems (to ease the estimation)

- ❑ loops with finite bounds
- ❑ no recursion
- ❑ no dynamic memory allocation
- ❑ no goto statements
- ❑ single-threaded if possible

Methods

Measuring:

- Compile, link and download onto target CPU
- Hook up logic analyzer or oscilloscope or use built in registers
- Run the code with test inputs, and record execution times
- Take the maximum as WCET
- No guarantee to hit the worst case!

Simulation:

- Various levels of precision possible (cycle accurate, instruction accurate); difficulty to simulate behavior of environment.

Analysis:

- Compute estimate of run time, based on program analysis and model of target hardware

Overview of the methods results

WCET: The longest time taken by a software task to execute

=> Function of input data and environment conditions

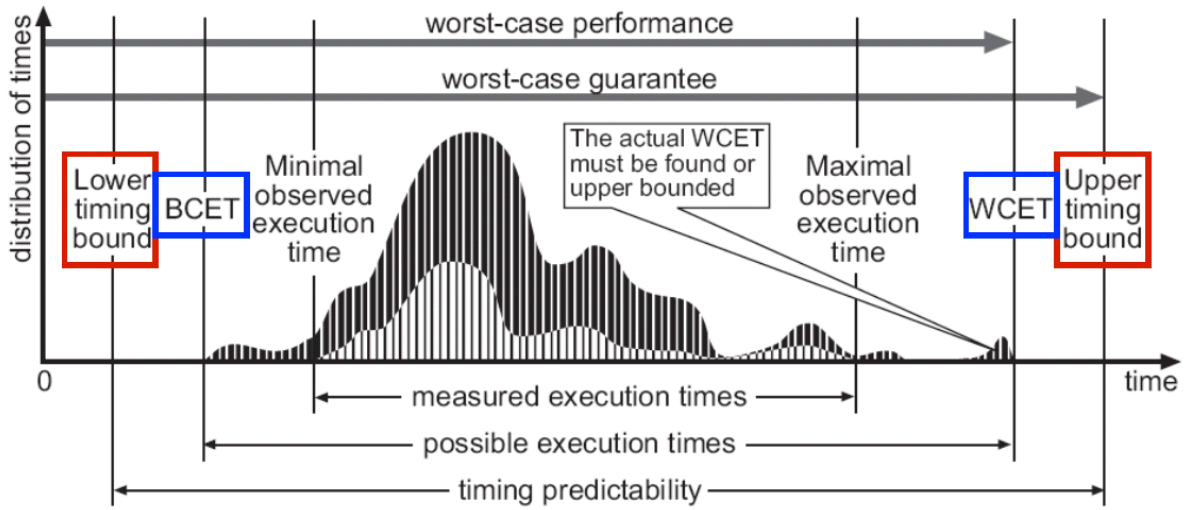


Figure from R.Wilhelm et al., ACM Trans. Embed. Comput. Sys, 2007.

Influence of input size

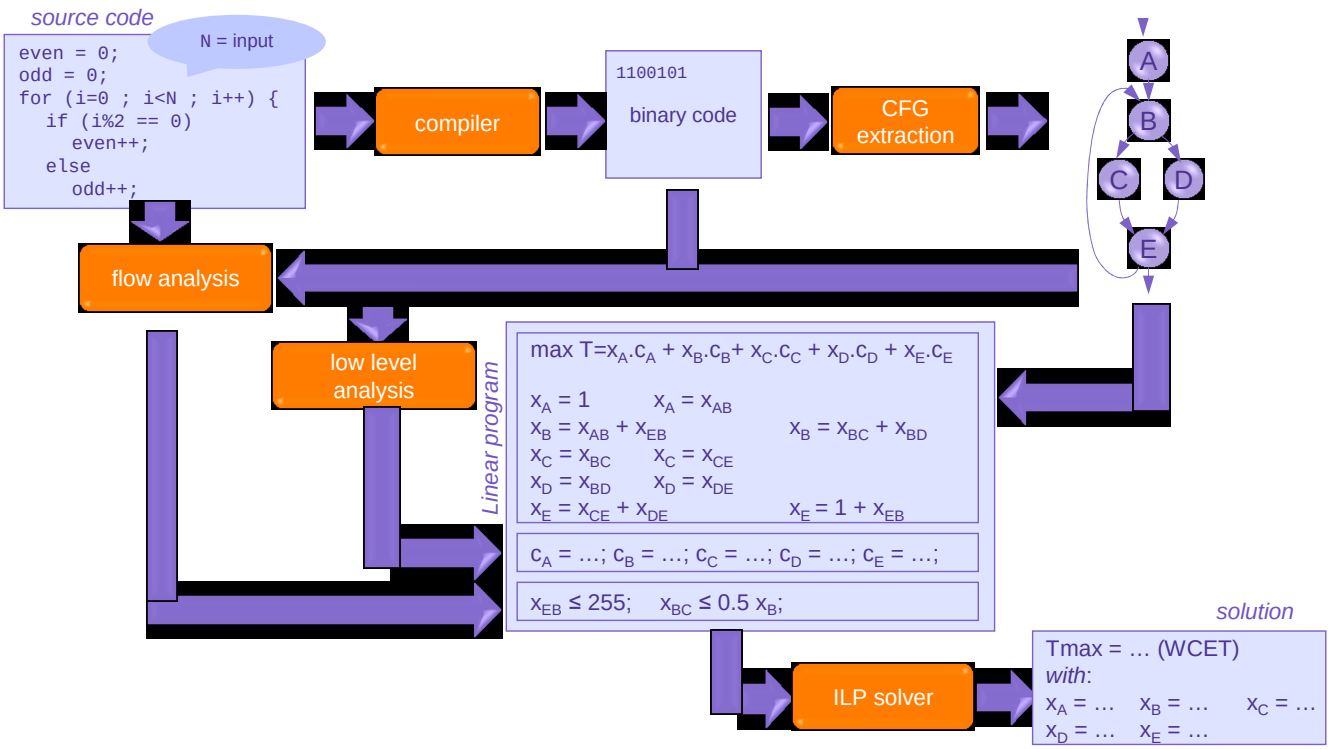
- **Instruction execution time can be different**
  - MUL (multiply) can take a variable # clock cycles, depending on input size

- **Control flow (while, for loops) can run for variable number of iterations**

Parameters:
• A positive integer, n.
Returns: The sum of the integers from 1 to n.

```
{ sum := 0;
  for i := 1 upto n
  { sum := sum + i;}
return sum;}
```


WCET computation by static analysis



[Rochange2012]

Control flow graph (CFG)

- **Nodes represent basic blocks. A Basic Block (BB) is piece of code with a single entry point and a single exit point, with no branching in-between.**
- **Edges represent flow of control (jumps, branches, calls,...)**

Example

test.c

```
int ex(int n) {
  int i, s=0;
  for(i=0; i< n; i++){
     s+=i;
  }
return s;
}
```

terminal

```
>gcc –c –g –lrm test.c
>dissy test.o
```

Deassemble the code

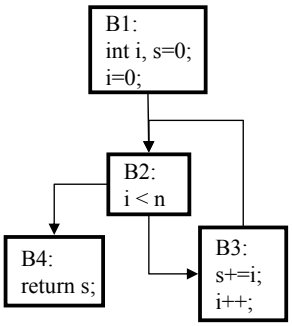
```
0x0000005e    push   %ebp
0x0000005f    mov    %esp,%ebp
0x00000061    sub    $0x10,%esp
0x00000064    movl   $0x0,-0x8(%ebp)        BB1
0x0000006b    movl   $0x0,-0x4(%ebp)
0x00000072    movl   $0x1,-0x4(%ebp)
0x00000079    jmp    85
0x0000007b    mov    -0x4(%ebp),%eax
0x0000007e    add    %eax,-0x8(%ebp)        BB3
0x00000081    addl   $0x1,-0x4(%ebp)
0x00000085    mov    -0x4(%ebp),%eax
0x00000088    cmp    0x8(%ebp),%eax         BB2
0x0000008b    jl     7b
0x0000008d    mov    -0x8(%ebp),%eax
0x00000090    leave                         BB4
0x00000091    ret
```


CFG - example

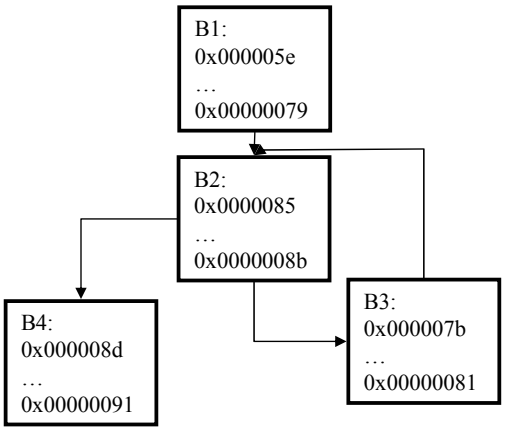
test.c

```
int ex(int n) {
  int i, s=0;
  for(i=0; i< n; i++){
     s+=i;
  }
return s;
}
```

CFG idea



Generated CFG



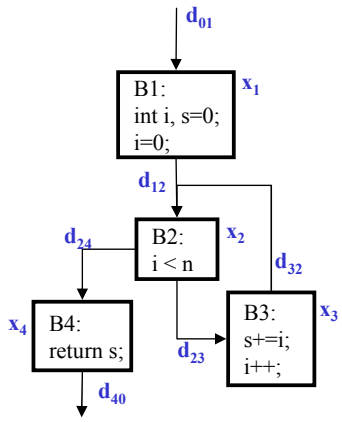
Identification of the longest path in a CFG

CFG can have loops, how to infer loop bounds?

- unroll loops, resulting in directed acyclic graph (DAG)
- construct system of equations

Example

$x_i \rightarrow$ # times Bi is executed
$d_{ij} \rightarrow$ # times edge is executed
$C_i \rightarrow$ WCET of B_i



maximize $\sum_i C_i x_i$

subject to flow constraints

$d_{01} = 1$
$x_1 = d_{01} = d_{12}$
$x_2 = d_{12}+d_{32} = d_{24}+d_{23}$
$x_3 = d_{23} = d_{32} = n$
$x_4 = d_{24} = d_{40}=1$

designer must give an upper bound of n

Computation of basic blocks WCET

Require to make a micro-architecture modeling

- perform a cycle-wise evolution of the pipeline, determining all possible successor pipeline states

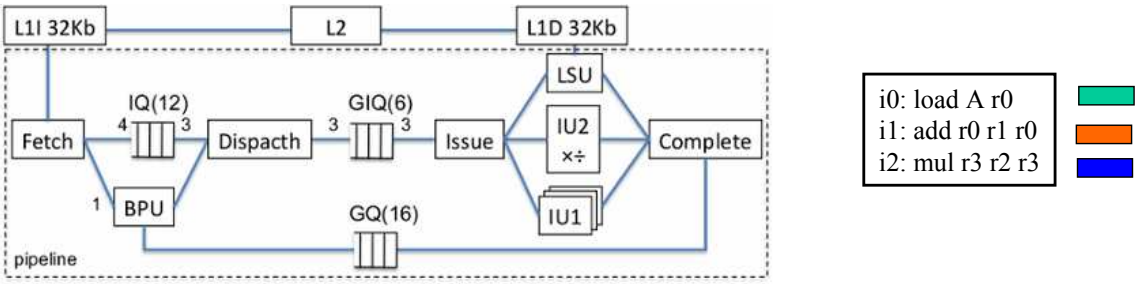
Model of the behaviour of the architecture

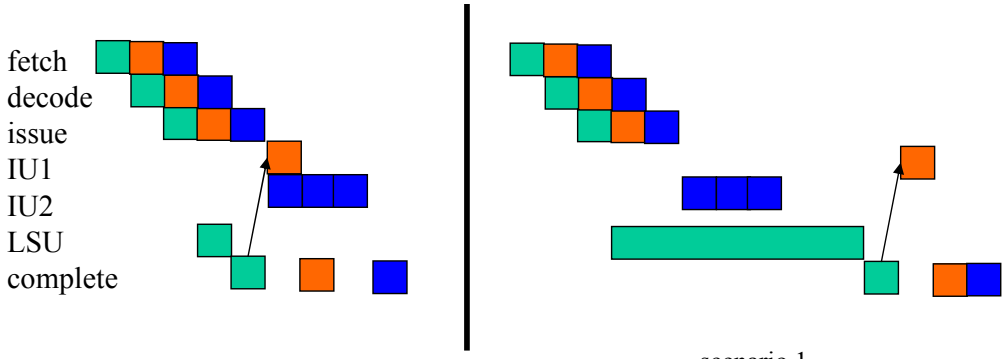
- Pipeline
  - Determine each instruction's worst case effective execution time by looking at its surrounding instructions within the same basic block.
- branch prediction
  - Predict which branch to fetch (ex static, always then for an if)
- Data dependent instruction execution times
- caches
- in case of multicore, common bus
- …

Read the processor handbook

Example - micro architectural analysis

Let us consider the following processor architecture and basic block





scenario 1
A in the L1D

scenario 1
A not in L1 and not in L2

Partie I - Introduction générale aux systèmes temps réel

1. **Système temps réel**

2. **Architecture des systèmes temps réel**

3. **Calcul du WCET**
   1. Méthode d'analyse statique
   2. Implantation prédictible sur multi-cœurs
   3. Et la suite?

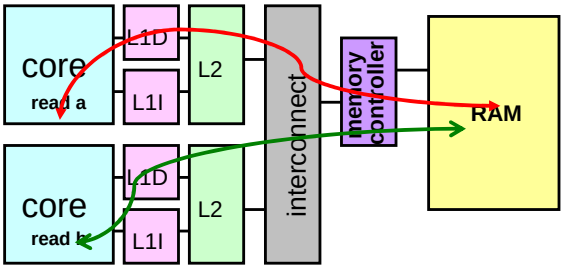
4. **Problématiques de conception des systèmes temps réel**

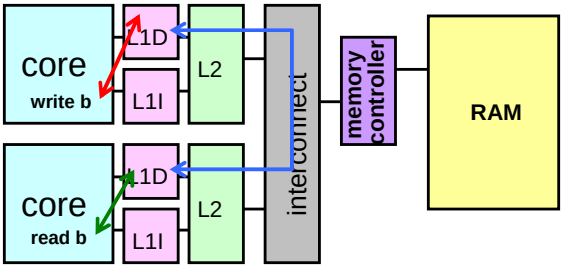
5. **Un peu de POSIX**

WCET for multicore COTS

Multicore COTS are unpredictable

1. the shared internal bus access
2. the cache coherency
3. limited available documentation
4. not enough monitoring applications
5. No *timing-composability* on COTS





Current solution

Use of an *execution model*

– Set of rules to avoid or reduce the unpredictable behaviours

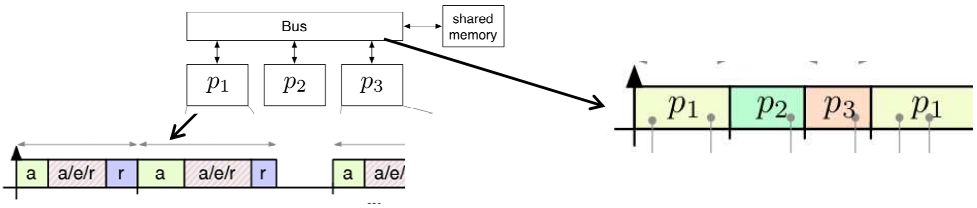
Develop a *bare-metal library*

– **Bare-metal:** low-level programming method for high-performance and real-time computing
  - bypass OS
  - minimal footprint

– Implementation of the execution model rules

⇒Idea is to reduce the problem to a well known problem (uni-processor case)

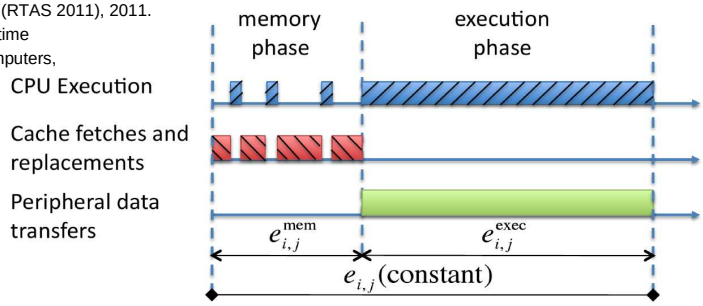
A. Schranzhofer, J.-J. Chen, and L. Thiele, "Timing analysis for TDMA arbitration in resource sharing systems," in 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2010), Stockholm, Sweden, 2010.



Current solutions

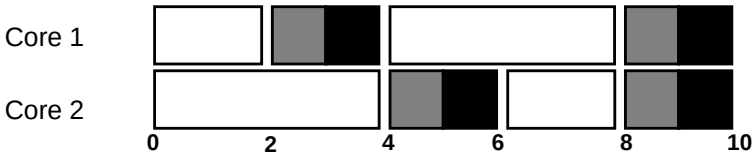
PREM (predictable execution model)

- R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley.A predictable execution model for COTS-based embedded systems. In 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2011), 2011.
- E. Betti, S. Bak, R. Pellizzoni, M. Caccamo, and L. Sha. Real-time i/o management system with cots peripherals. IEEE Trans. Computers, 62(1):45–58, 2013.



Time-triggered execution model

F. Boniol, H. Cassé, E. Noulard, C. Pagetti: Deterministic Execution Model on COTS Hardware. In 25th International Conference on Architecture of Computing Systems (ARCS 2012), 2012.



Exemple d'implantation prédictible sur multicoeur

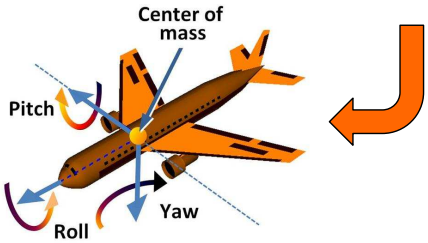
Avionic use case: Flight Management System (FMS)

Computer system specific to civil & military avionic responsible of:

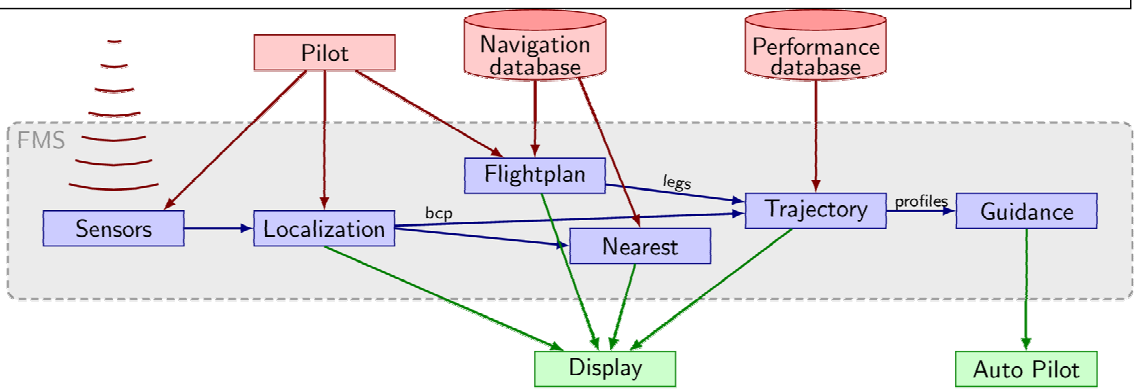
- Management of the flight plan
- Localization (through various sensors & radio navigation)
- Trajectory Computation along the flight plan
- Translation into Guidance Information for the autopilot

– Strong safety-critical constraints

- Hard real-time requirements
- Periodic and Aperiodic tasks



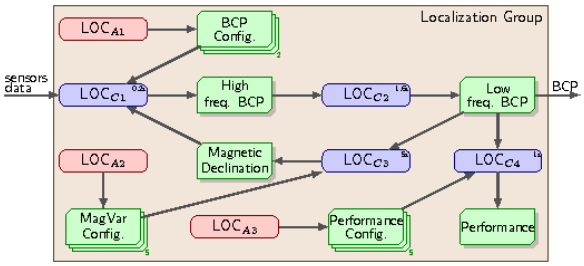
FMS Software Architecture



Functional groups	Description	Periodic	Aperiodic
Sensors	**Gather sensor data from devices**	**1**	**4**
Localization	**Estimate the airplane location from sensors**	**4**	**3**
Flightplan	**Deals with flight route the plane should follow**	**0**	**7**
Trajectory	**Compute a trajectory to tangent the route**	**5**	**0**
Guidance	**Translate trajectory into angle modifications**	**2**	**0**
Nearest	**List nearest airports for emergency landing**	**1**	**0**

Localization group

- **Communication:**
  - Blackboard: single-writer / multiple-reader double-buffer
  - Mailbox: single-writer / single-reader FIFO used to enqueue pilot and co-pilot requests



- **Real-time constraints:**
  - Periodic task: T period
  - Aperiodic task: max activation

Task	Period
LocC1	**200ms**
LocC2	**1.6s**
LocC3	**5s**
LocC4	**1s**

Task	Max activation
LocA1	**2 / 200ms**
LocA2	**5 / 5s**
LocA3	**5 / 1s**

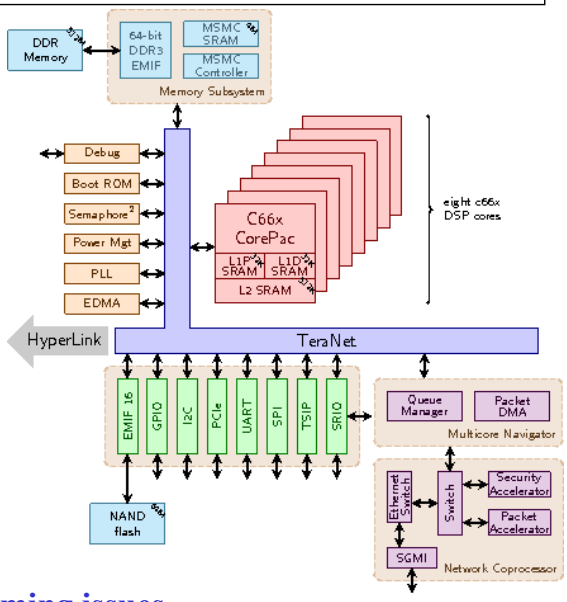
- **Memory footprint:**

Task	Code size (Bytes)	Data size	Output size
LocC1	**6236**	**2744**	**1232**
LocC2	**3072**	**1360**	**1232**
LocC3	**3892**	**1104**	**272**
LocC4	**2528**	**1104**	**208**

Task	Code size (Bytes)	Data size	Output size
LocA1	**9012**	**200**	**136**
LocA2	**9216**	**224**	**328**
LocA3	**9524**	**224**	**328**

Texas Instruments TMS320C6678

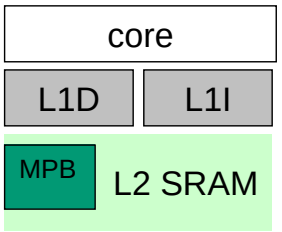
- **8 TMS320C66x DSP processors**
  - clocked at 1 GHz
  - VLIW instruction set architecture
  - 32 KB program memory (L1P)
  - 32 KB data memory (L1D)
  - 512 KB level 2 memory (L2)
- **Several platform configurations**
  - L1P, L1D, and L2 memories can be configured as cache, SRAM, or a mix of both
  - two different addresses: global address (accessible by all cores) or local address.
    - E.g.: core 3. Local address = 0x00800000 or global address = 0x13800000



- **Timing issues**
  - local time-stamp counter
  - cores start independently => unpredictable offsets
  - Debugger may interrupt the core (e.g. printf)

Platform configuration

Rule 1: static platform memory configuration



L1D & L1P: cache

L2: 100% SRAM

.stack, .text, .data, .heap ➔ L2 SRAM

MPB (Message Passing Buffer) :

communication zone

blackboard, mailbox ➔ MPB

DDR:

Statically mapped

Navigation data-base ➔ DDR

Pros:

Spatial partitioning: cores run in isolation, except when accessing the MPB or the DDR

Cons:

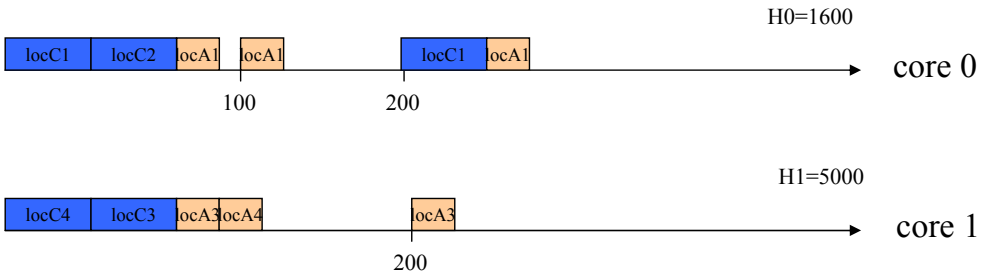
Task must fit in the L2

Scheduling strategy

Rule 2: static dispatcher on each core

Partitioned non preemptive schedules

Static schedule



Pros:

Temporal partitioning: tasks run in isolation, except when accessing the MPB or the DDR

Cons:

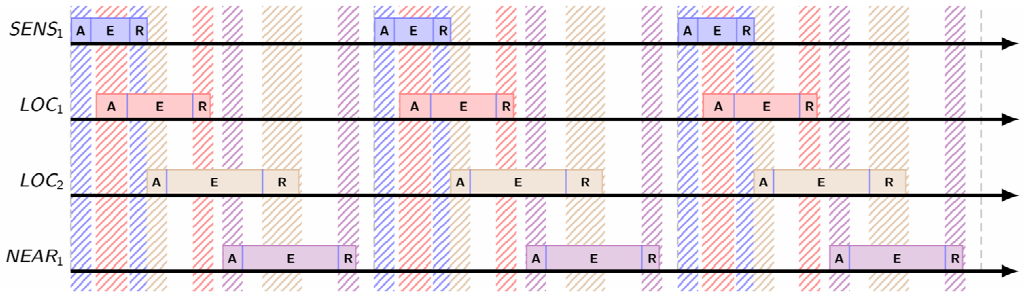
Not flexible

Super-block decomposition: AER model

Rule 3: super-block decomposition

Acquisition – Execution - Restitution (AER) execution model

3 phases: acquisition (copy input data locally), execution (local execution), and restitution (copy output data from local to distant memory)



Pros:

Execution in isolation: no concurrency when accessing MPB and DDR

Cons:

Specific programming rules

Complexity for finding a valid dispatch

Partie I - Introduction générale aux systèmes temps réel

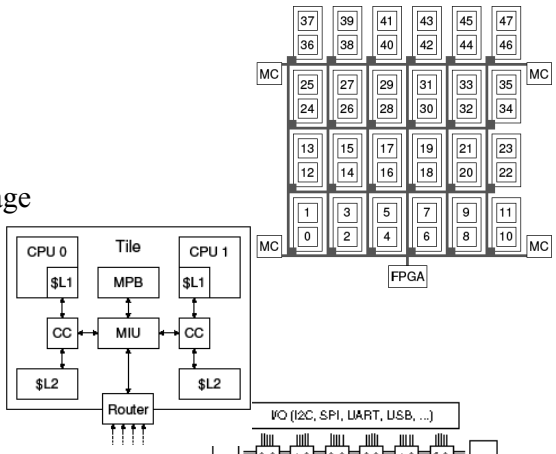
1. Système temps réel
2. Architecture des systèmes temps réel
3. Calcul du WCET
   1. Méthode d'analyse statique
   2. Implantation prédictible sur multi-cœurs
   3. Et la suite?
4. Problématiques de conception des systèmes temps réel
5. Un peu de POSIX

Many-core

Arrivée des many-core

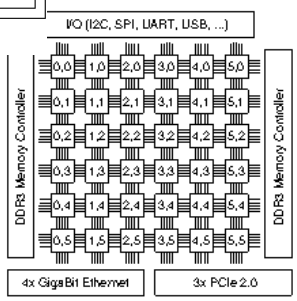
Intel SCC (Single-chip Cloud Computer)

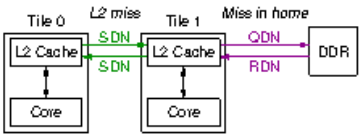
- 24 dual-core tiles répartis en grille 6×4
- 48 coeurs, dérivés anciennes familles Pentiums
- Chaque tile contient un router et 16 KB pour passage de message (MPB)
- Cœurs ne dérivent pas mais démarrent avec offset



TILEmpower-Gx36

- 36 tiles répartis en grille 6×6
- Cœurs ne dérivent pas mais démarrent avec offset
- Cohérence de mémoire partagée avec politique homing

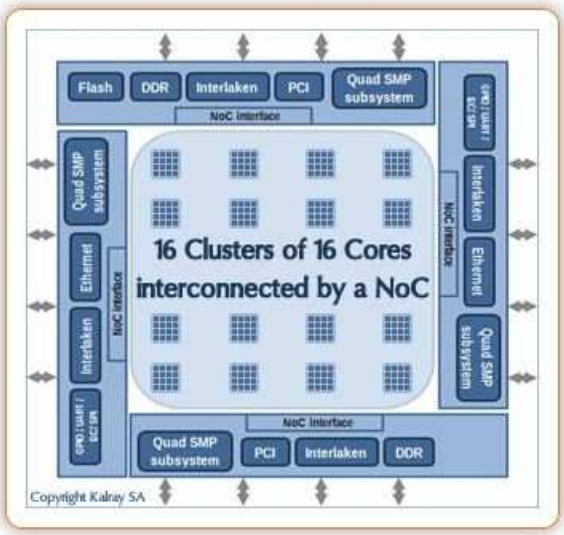




Arrivée des many-core

Kalray MPPA

– 256 cœurs



MPPA® block diagram

Plan

1. Systèmes temps réel
2. Architecture des systèmes temps réel
3. **Problématiques de conception de systèmes temps réel**
4. Un peu de POSIX

Caractéristiques

Programme « standard »:

1. (1) termine; (2) retourne un résultat et (3) manipule des données complexes mais sa structure de contrôle est assez simple.
2. Pour ces programmes standards, les propriétés à prouver sont toujours du style “quand la fonction est appelée et que la précondition est vérifiée, alors la fonction termine et la postcondition est vérifiée”.

Exemple typique de programme standard : compilateur, algorithme de tri.

Programmes réactifs temps réel.

1. (1) ils ne terminent pas forcément; (2) ils ne calculent pas un résultat mais plutôt maintiennent une interaction ; (3) les types de données manipulés sont souvent simples alors que le contrôle est complexe (exécution de plusieurs composants en parallèle). Enfin bien souvent ils interagissent avec un environnement par le biais de capteurs (prise d’information) et d’actionneurs (action).

[Bar08]

Propriétés temporelles

Les propriétés que l’on veut prouver sur ces systèmes réactifs sont très différentes de celles que l’on veut prouver sur des programmes standards. On veut typiquement prouver des propriétés sur l’entrelacement des événements tout au long de l’exécution (infinie) du programme, par exemple

-si un processus demande infiniment souvent à être exécuté, alors l’OS finira par l’exécuter;

- il est toujours possible lors de l’exécution de revenir à l’état initial;

- chaque fois qu’une panne est détectée, une alarme est émise;

- chaque fois qu’une alarme a été émise, une panne avait été détectée.

Schématiquement, pour les programmes standards, les propriétés à vérifier impliquent des prédicats très riches sur les données manipulées (ex. le tableau doit être trié) mais les aspects temporel sont très restreints, tandis que pour les systèmes réactifs l’aspect temporel est très élaboré mais les prédicats sur les données sont souvent basiques (ex. x!=0).

[Bar08]

Quels sont les choix pour le concepteur?

1. **Architecture matérielle:** au niveau système et composant. L'architecture du système doit prendre en compte tous les types de contrainte, y compris des contraintes de distance physique ou de poids. Le composant faisant partie du tout doit se charger de ses entrées / sorties, de son interface et des moyens d'intreconnexion.
2. **Modèle de communication:** modèle interne de communication entre processus d'un composant et modèle de communication entre composants. Plusieurs solutions: envoi de messages, rendez-vous, … Choix du support physique: time triggered (1553, Arinc429, TTP…), temps réel (CAN, AFDX…), classique (Ethernet, Wifi…)
3. **Technologie des calculateur:** PC, terminal, processeur, SOC, …
4. **Operating system et ordnnancement:** operating system temps réel (VxWorks, PikeOS, Arinc 653,…), micro kernel, pas OS …
5. **Formalismes et langages d'aide au développement:** programmation à la main (C, Ada,…), langages semi formel de haut niveau (AADL, UML…), langages et formalismes formels de haut niveau (SCADE, Lustre, SDL, automates temporisés…)

Quels sont les choix pour le concepteur?

1. **Architecture matérielle:** au niveau système et composant. L'architecture du système doit prendre en compte tous les types de contrainte, y compris des contraintes de distance physique ou de poids. Le composant faisant partie du tout doit se charger de ses entrées / sorties, de son interface et des moyens d'intreconnexion.
2. **Modèle de communication:** modèle interne de communication entre processus d'un composant et modèle de communication entre composants. Plusieurs solutions: envoi de messages, rendez-vous, … Choix du support physique: time triggered (1553, Arinc429, TTP…), temps réel (CAN, AFDX…), classique (Ethernet, Wifi…)
3. **Technologie des calculateur:** PC, terminal, processeur, SOC, …
4. **Operating system et ordnnancement:** operating system temps réel (VxWorks, PikeOS, Arinc 653,…), micro kernel, pas OS …
5. **Formalismes et langages d'aide au développement:** programmation à la main (C, Ada,…), langages semi formel de haut niveau (AADL, UML…), langages et formalismes formels de haut niveau (SCADE, Lustre, SDL, automates temporisés…)

Plan

1. Systèmes temps réel
2. Architecture des systèmes temps réel
3. Problématiques de conception de systèmes temps réel
4. Un peu de POSIX

Standard POSIX

POSIX

– Portable Operating System Interface [for Unix]

– name of a family of related standards specified by the IEEE to define the application programming interface (API), along with shell and utilities interfaces for software compatible with variants of the Unix operating system, although the standard can apply to any operating system

Advantages:

– COTS often implement some POSIX functionalities

– portability: a compliant POSIX application should be ported on several systems

RTOS compliant with POSIX:

–VxWorks (partially)

– QNX

– RTEMS (Real-Time Executive for Multiprocessor Systems), free open source real-time operating system designed for embedded systems.

– RTAI

– RTLinux (partially)

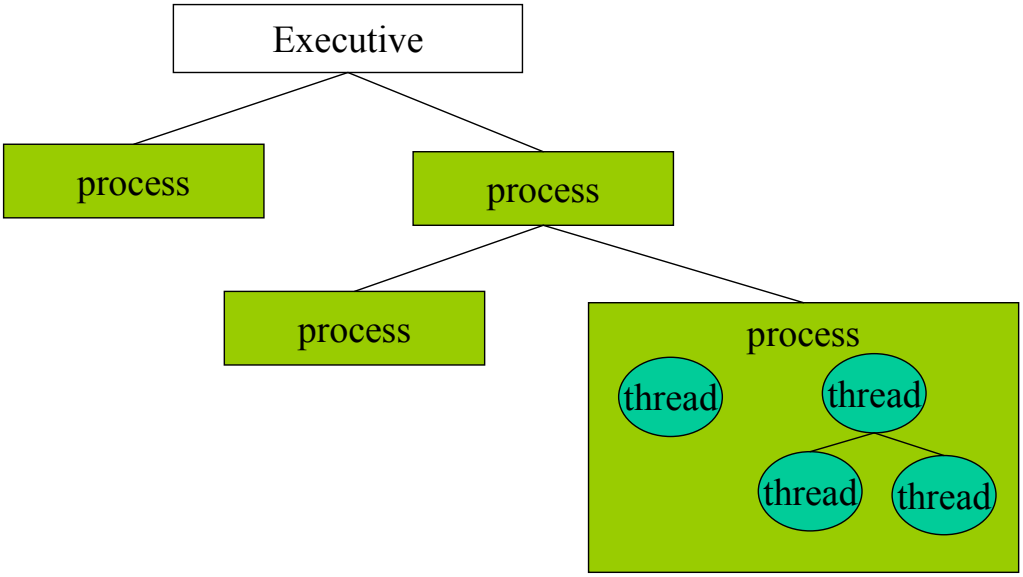
– Marte OS (subset)

– …

Process and task in POSIX

Hierarchy of processes and tasks

- Process = entity with its own memory
- Thread = internal entity of a process. The threads of a process share the memory



Brief overview of POSIX

POSIX.1 **Core Services** **IEEE Std 1003.1**	**Process Creation and Control, Signals, Timers ...**	**Define API with the RTOS**
POSIX.1b **Real-time extensions** **IEEE Std 1003.1b**	**Priority Scheduling, Real-Time Signals, Clocks and Timers, Semaphores, Message Passing …**	**Define the extensions for real-time**
POSIX.1c **Threads extensions** **IEEE Std 1003.1c**	**Thread Creation, Control, and Cleanup; Thread Scheduling …**	**Define the API for the threads management**

Functions on thread

boussens.cert.fr 46 ==: man -k pthread

pthread.h [] (0p) - threads

pthread_atfork [] (3p) - register fork handlers

pthread_attr_destroy [] (3) - initialize and destroy thread attributes object

pthread_attr_destroy [] (3p) - destroy and initialize the thread attributes object

pthread_attr_getaffinity_np [] (3) - set/get CPU affinity attribute in thread attributes object

pthread_attr_getdetachstate [] (3) - set/get detach state attribute in thread attributes object

pthread_attr_getdetachstate [] (3p) - get and set the detachstate attribute

pthread_attr_getguardsize [] (3) - set/get guard size attribute in thread attributes object

pthread_attr_getguardsize [] (3p) - get and set the thread guardsize attribute

pthread_attr_getinheritsched [] (3) - set/get inherit scheduler attribute in thread attributes object

pthread_attr_getinheritsched [] (3p) - get and set the inheritsched attribute (REALTIME THREADS)

pthread_attr_getschedparam [] (3) - set/get scheduling parameter attributes in thread attributes object

…

Example: man pthread_create

pthread_create - create a new thread

–#include <pthread.h>

–**synopsis:** int pthread_create(pthread_t * thread, pthread_attr_t * attr, void * (*start_routine)(void *), void * arg);

–pthread_create() creates a real-time thread that will have attributes given by **attr**, and that begins executing function **start_routine(arg).** If the attribute is NULL, default attributes are used. See **pthread_attr_init** for a complete list of attributes. Upon successful completion, **pthread_create()** shall store the ID of the created thread in the location referenced by thread.

If the **start_routine** returns, the effect shall be as if there was an implicit call to **pthread_exit()** using the return value of start_routine as the exit status.

If successful, the **pthread_create()** function shall return zero; otherwise, an error number shall be returned to indicate the error.

Bibliographie

1. Frédéric Boniol: cours sur les systèmes temps réem
2. Pierre-Emmanuel Hladik: cours sur les systèmes temps réel
3. Jean-Pierre Elloy: cours sur les systèmes temps réel
4. Gilles Geeraerts: Introduction aux systèmes embarqués
5. Wikipedia
6. Pascal Brisset. Paparazzi. http://www.recherche.enac.fr/~brisset/2010-02_ONERA_paparazzi.pdf
7. Stéphane Bardin. Introduction au Model Checking, 2008. http://sebastien.bardin.free.fr/MC-ENSTA.pdf.

Plan

1. Partie I - Introduction générale aux systèmes temps réel et aux problématiques de conception
2. Partie II - SDL
3. Partie III – Langage Lustre: syntaxe, sémantique et preuve
4. Partie IV – Introduction aux automates temporisés

Organisation du cours

1. **Présentation de Lustre**
   1. Syntaxe
   2. Sur-échantillonnage et sous-échantillonnage

2. **Quelques points sémantiques**

Approche

Flot de données (data-flow):

- X est une suite infinie de valeurs X_n avec $n \geq 0$
- X_n est la n-ième valeur de X

Définition de flot:

- Un flot est défini par une équation graphique O=F(X,Y,..) qui permet de calculer O_n en fonction de X_n et Y_n (à chaque instant)

Un programme SCADE:

- Ensemble d'équations
- Description modulaire
- A chaque instant, les valeurs des flots sont calculées en fonctions des valeurs des flots d'entrée

Syntaxe générale

```
[declaration of types and external functions]
node name (declaration of input flows)
returns (declaration of output flows)
[var declaration of local flows]
let
  [assertions]
  system of equations defining once each local flow
  and output depending on them and the inputs
tel.

[other nodes]
```

Types :

- basic types: int, bool, real
- tabular : int^3, real^5^2…

Operators

Classical operators:

Arithmetical :

Binary : `+`, `-`, `*`, `div`, `mod`, `/`, `**`

Unary : `-`

Logical :

Binary : `or, xor, and, =>`

Unary : `not`

Comparison :

`=`, `<>`, `<`, `>`, `<=`, `>=`

Control :

`if . then . else`

Temporal operators :

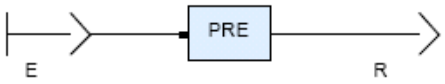
`pre` (precedent) : operator which allows to work on the past of a flow

`->` (followed by) : operator which allows to initiate a flow

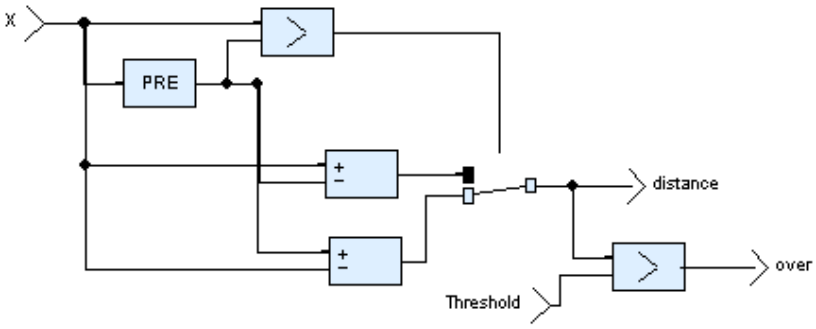
Opérateur pre - rappel

Mémorise les valeurs précédentes du flot

- Flot X = (X_0, X_1, … , X_n, …)
- Flot `pre(X)`=(nil, X_0, X_1, … , X_n, …)



Exemple : détection de dépassement de seuil

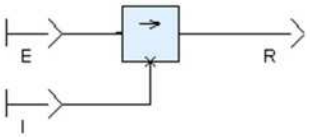



```
Distance = if (X>pre(X)) then X-pre(X) else pre(X)-X ;
Over = (Distance > Threshold) ;
```

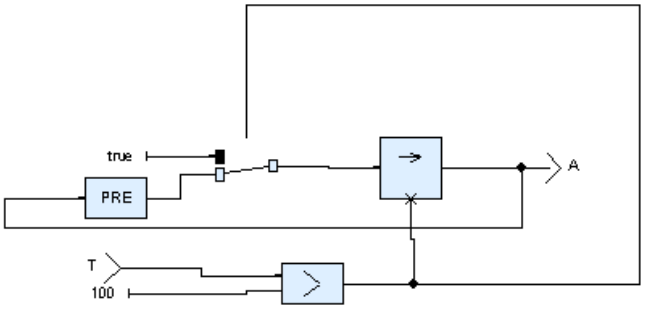

Operator -> - rappel

Initialise un flot

- Flots X= (X_0, X_1, … , X_n, …) et Y=(Y_0, Y_1, … , Y_n, …)
- `Y->X` = (Y_0, X_1, … , X_n, …)



Exemple : surveillance température




```
A = (T>100) -> if (T>100) then true else pre(A) ;
```

Exemples standards - rappel

A resetable counter

- Input: reset — reset the counter (Boolean flow)
- Output: counter — value of the counter (Integer flow)

Write the program

```
node a_counter (reset : bool)
returns (counter : int)
let
    counter = 0-> if reset then 0 else pre counter +1;
tel

⇔ counter = if reset then 0 else (0-> pre counter +1);
```


Exercise

A timer:

- Input: set — activation of the timer (Boolean flow)
- Output: level — state of the timer (Boolean flow)
- Constante: delay — duration of the timer in number of tops

Write the program

```
const delay : int;
node timer (set : bool)
returns (level : bool)
var c : int;
let
   c = if set then 0 else (N+1-> pre c +1):
   level = (c=N);
tel
```

Organisation du cours

1. **Présentation de Lustre**
   1. Syntaxe
   2. Sur-échantillonnage et sous-échantillonnage

2. **Quelques points sémantiques**

Clock-based semantics

Clock:

- a clock is a Boolean flow

Basic clock:

- is the flow true

Semantics of a flow:

- is the sequence of pairs (v_i,c_i) where v_i is the value and c_i is the clock associated to the flow

Equation:

- must be homogenous in term of clock
  - X + Y has a sense iff X and Y have the same clock

Operator when

Sub-samples a flow on a lower clock

Let X be a flow and B a Boolean flow (assimilated to a clock) of same clock.
The equation

```
Y = X when B
```

defines a flow Y, of same type than X, and of clock B

- Y is present when B is true
- Y is absent when B is false or when B and X are absent

Operator current

Over-samples a flow on a quicker clock

Let X be a flow and B a Boolean flow (assimilated to a clock) of same clock.
The equation

```
Y = X current B
```

defines a flow Y, of same type than X, and of clock the clock of B

- Y is present iff B is present
- when Y is present, Y is equal to X if X is present and to the previous value of X otherwise

Sampling

The operator when defines a *slower* flow than the input

X	4	1	-3	0	2	7	8
C	*true*	*false*	*false*	*true*	*true*	*false*	*true*
X when C	4			0	2		8

Question: is the flow ***X + X when C*** well defined?

The operator current constructs a *faster* flow than the input

X	4	1	-3	0	2	7	8
C	*true*	*false*	*false*	*true*	*true*	*false*	*true*
Y=X when C	4			0	2		8
current (Y)	4	4	4	0	2	2	8

Warning: current (X when C) ≠X

Problem of initialisation of current

X	4	1	-3	0	2	7	8
C	*false*	*false*	*false*	*true*	*true*	*false*	*true*
current(X when C)	nil	nil	nil	0	2	2	8

Idea 1 : sample with clocks which are initially true

C1 = true -> C	*true*	*false*	*false*	*true*	*true*	*false*	*true*
current(X when C1)	4	4	4	0	2	2	8

Idea 2: force a default value

E = if C then current(X when C) else (dft -> pre E);

Idea 3: (to avoir additionnal memory)

X1 = (if C then X else dft) -> X;

E = current(X1 when C1);

Exercise

For the following counters:

```
node counter (reset : bool) returns (n : int)
let n = 0 -> if reset then 0 else pre n + 1; tel

node decounter (reset : bool) returns (n : int)
let n = 0 -> if reset then 0 else pre n - 1; tel

node counter2 (reset,c : bool) returns (n : int)
let n = if c then counter(reset)
            else decounter(reset); tel
```

Precise the behaviour of the node

```
node counter3 (reset,c : bool) returns (n : int)
var c1,c2 : int;
let c1 = current (counter(reset when c));
    c2 = current (counter2 (reset when not c), (c when not
  c));
     n = if c then c1 else c2; tel
```


Recapitulative

B	false	true	false	true	false	false	true	true
X	x0	x1	x2	x3	x4	x5	x6	x7
Y	y0	y1	y2	y3	y4	y5	y6	y7
pre(X)	nil	x0	x1	x2	x3	x4	x5	x6
Y->pre(X)	y0	x0	x1	x2	x3	x4	x5	x6
Z=X when B		x1		x3			x6	x7
T=current Z	nil	x1	x1	x3	x3	x3	x6	x7
pre(Z)		nil		x1			x3	x6
0->pre(Z)		0		x1			x3	x6

Exercice

Ecrire le comportement du programme

```
node exo(X: int; C,D : bool;) returns(Y : bool)
var Z : int when C;
let
  Z = current ((X when C) when D);
  Y= current(Z);
tel
```


Exercice: Additionneur 3 bits

Les booléens sont interprétés comme des chiffres binaires:

- **faux = 0, vrai = 1**

Ecrire un nœud (combinatoire)

- **node Add3b(cin, x, y : bool) returns (cout, s : bool);**
- **qui réalise l'addition binaire de cin, x et y**
  i.e., pour tout t $cin_t + x_t + y_t = 2 * cout_t + s_t$

Exercice: Additionneur série

Les flots de booléens sont interprétés (quand c'est possible) comme des nombres binaires "arbitrairement long" :

$$X = X_0 + X_1 * 2 + X_2 * 2^2 + ... + X_t * 2^t + ...$$

1. **Quel est l'entier représenté par le flot *false* ?**
2. **Quel est l'entier représenté par le flot *true -> false* ?**
3. **Ecrire un additionneur série:**
   1. **node AddSerie(X, Y : bool) returns (S : bool);**
   2. **tel que le flot S représente la somme des nombres binaire X et Y.**
4. **Que vaut le flot *AddSerie(true->false, true->false)* ?**
5. **Comment peut-on interpréter le flot *true* ?**

Organisation du cours

1. **Présentation de Lustre**
   1. Syntaxe
   2. Sur-échantillonnage et sous-échantillonnage

2. **Quelques points sémantiques**
   1. Sémantique
   2. Compilation
   3. Traduction automates Scade

Sémantique de traces

Notations: X est un flot x est une valeur de flot et nil est l'absence de valeur

Constantes:

- 0 = 0.0

Opérateurs usuels:

- nil.X op nil.Y= nil.X op Y
- x.X op y.Y = (x op y).X op Y

Valeur initiale:

- nil.X -> nil.Y = nil. X -> Y
- x.X -> y.Y = x.Y

Sémantique de traces

Retard:

- pre nil.X = nil.pre X
- pre x.X = nil.pre# x X
- pre# x x'.X = x.pre# x' X
- pre# x nil.X = nil.pre# x X

Sous-échantillonnage:

- nil.X when nil.C= nil.X when C
- x.X when vrai.C = x.X when C
- x.X when false.C = nil.X when C

Sémantique de traces

Sur-échantillonnage:

- current nil.X nil.C = nil. current X C
- current nil.X false.C = nil. current X C
- current x.X true.C = x. current# x X C
- current# x x'.X vrai.C =x'.current# x' X C
- current# x nil.X faux.C = x.current# x X C
- current# x nil.X nil.C = nil.current# x X C

Exercice

Donner la sémantique de

- x<>y
- if C then x else y

Sémantique de traces

Un nœud est une fonction qui associe aux sorties la solution du système d'équations appliqué aux entrées.

Les équations d'un nœud Lustre forment une fonction

$$f : D^{\infty} \times D^{\infty} \times D^{\infty} \rightarrow D^{\infty} \times D^{\infty}$$

tel que (s,v) = f(e,s,v) où s sont les sorties, e les entrées et v les variables intermédiaires.

Le nœud complet réalise la fonction (solution de l'équation X=F(X))

$$e \mapsto \mu_{s,v} f(e,s,v)$$

où est $\mu_{s,v}$ le plus petit point fixe de f par rapport à ses composantes s et v.

Sémantique de traces

Une manière effective de calculer ce point fixe est

$$\mu_{s,v} f(e,s,v) = \sup_n \{f^n(e, \varepsilon, \varepsilon)\}$$

Ainsi la solution consiste à calculer de manière itérative

$$f(\varepsilon, \varepsilon, \varepsilon)$$
$$f(e_0, f(\varepsilon, \varepsilon, \varepsilon))$$
$$f(e_0.e_1, f(e_0, f(\varepsilon, \varepsilon, \varepsilon)))$$
$$\ldots$$

Cela revient au principe d'exécution:

initialisations
tant que vrai
    lire les entrées
    calculer les sorties
    mettre à jour les mémoires
fin tant que

Les horloges

- **L'horloge de base est vrai,vrai,vrai,vrai…=vrai$^\omega$**
- **Les constantes sont sur l'horloge de base**
  - 2 ≡ (2,vrai),(2,vrai), (2,vrai), (2,vrai), …
- **Les entrées provenant de l'environnement sont sur l'horloge de base**

- **L'horloge de *Z = X op Y* est $h_Z = h_X = h_Y$**
  - X = 2 + Y alors $h_X = h_2 = h_Y$ (= vrai$^\omega$ si Y est une entrée)
- **L'horloge de *Z = X -> Y* est $h_Z = h_X = h_Y$**
- **L'horloge de *Z = pre X* est h_Z coïncide avec h_X après la 2ème valeur et vaut nil avant**
  - X = pre 2 alors h_X = faux.vrai$^\omega$

- **L'horloge de *Z = X when C* est $h_Z = h_X$ on C**
  - X = 2 when C alors h_X = C si h_c = vrai
- **L'horloge de *Z = current (X when C)* est $h_Z = h_X = h_C$**

Hiérarchie d'horloges

- **L'imbrication des when forme un arbre d'horloges dont la racine est l'horloge de base du nœud.**

- **L'horloge de base d'un nœud est l'horloge de la plus rapide de ses entrées.**

- **Tandis que le when crée une nouvelle horloge (horloge fille) dans l'arbre d'horloges, le current permet de remonter d'un niveau et un seul dans l'arbre (horloge mère).**
  - vrai.C1 on vrai.C2 = vrai.(C1 on C2)
  - vrai.C1 on false.C2 = false.(C1 on C2)
  - nil.C1 on nil.C2 = nil.(C1 on C2)

Organisation du cours

1. **Présentation de Lustre**
   1. Syntaxe
   2. Sur-échantillonnage et sous-échantillonnage

2. **Quelques points sémantiques**
   1. Sémantique
   2. Compilation
   3. Traduction automates Scade

Analyses statiques

Vérifier la correction du programme avant de générer du code.

Analyse de causalité:

– Vérifie l'absence de cycle dans les définitions de flots.

Analyse d'initialisation

– Vérifie que l'on n'accède pas à des valeurs non initialisées (pre).

Calcul d'horloges

– Vérifie que l'on ne combine que des flots d'horloges identiques.

Analyse de causalité

- **De nombreux langages de type schéma/bloc (ex, Scade, Simulink) imposent une condition de non rebouclage direct, i.e., tous les cycles doivent traverser un retard (registre).**

- **Vérifie l'absence de cycle dans les définitions de flots (s'apparente à la recherche de cycle dans un graphe).**

Calcul d'horloge

- **Le calcul d'horloge associe à chaque flot son (unique) horloge et vérifie la correction des horloges.**
  - X + (X when C) sera rejeté par le calcul d'horloge

- **vérifie que l'on ne combine que des flots d'horloges identiques (pas d'accès à des valeurs de flot absentes).**

- **équivalence des horloges purement syntaxique**

  C = true -> not pre C;

  C1 = true -> not pre C1;

  Y = X when C et Y1 = X when C1 ne sont pas vus comme ayant la même horloge

Approche par inférence d'horloge

- **Idée : reprendre les techniques d'inférence de types (classique en ML)**

- **Calcul d'horloge = système de types.**

- **Système de types = ensemble de règles d'inférence.**

Règles d'inférence

$$\frac{c \in Dom(E)}{E \vdash c : E(c)}$$

- **E est un environnement associant un type (ou une horloge) à une expression.**
- **E⊢ x : t est un jugement de type déclarant que x a le type t dans l'environnement E.**
- **Une règle d'inférence dit que si les prémisses de la règle (partie haute) sont satisfaites alors on peut en déduire que la conclusion (partie basse) est vraie.**
- **Ci-dessus on déclare qu'une constante c a pour type E(c) si elle fait partie du domaine de E.**
- **On dit ainsi que c est bien typée si elle est liée (déclarée) dans E.**

Type fonctionnel

+ : int → int →int

- **→ désigne un type fonctionnel.**
- **On déclare que + est une fonction, qui s'applique à deux entiers et renvoie un entier.**
- **On déclare aussi que l'application de + à un seul entier produit une nouvelle fonction, qui s'applique cette fois à un seul entier (ex : (3+)4).**

Type polymorphe

$$\text{id} : \forall \alpha, \alpha \rightarrow \alpha$$

- **On déclare que la fonction identité prend un paramètre de type α et renvoie une valeur de type α.**
- **Et ce, quel que soit α.**
- **Ainsi identité s'applique aussi bien à des entiers, qu'à des chaînes de caractères et même à des valeurs fonctionnelles.**
- **Par contre on renvoie à chaque fois une valeur du même type que le type du paramètre.**

Règle d'inférence d'horloge du +

$$E \vdash + : \forall \alpha, \alpha \rightarrow \alpha \rightarrow \alpha$$

- **L'addition prend deux flots de même horloge.**
- **Cette horloge est quelconque.**
- **Si les flots ne sont pas de même horloge, ils sont mal synchronisés, la règle ne peut être appliquée.**

Règle d'inférence d'horloge fonctionnelle

$$\frac{E, x : cl_1 \vdash e : cl_2}{E \vdash \lambda x.e : cl_1 \rightarrow cl_2}$$

- **λx.e est une fonction qui associe la valeur e au paramètre x (ex : λx.x + 3).**
- **Pour calculer l'horloge de cette expression, on rajoute x dans l'environnement avec son horloge. Intuitivement, cela revient à déclarer les paramètres de la fonction dans l'environnement.**
- **Dans cet environnement enrichi, on calcule ensuite l'horloge de e.**
- **L'horloge obtenue pour e est l'horloge de retour de la fonction.**
- **L'expression résultante a une horloge de type fonctionnel.**

Règle d'inférence d'horloge fonctionnelle

$$\frac{E \vdash f : cl_1 \rightarrow cl_2 \quad E \vdash x \;\; cl_1}{E \vdash f\, x : cl_2}$$

- **f x est l'application de la fonction f à x.**
- **Pour que f x soit bien synchronisé, il faut que :**
  - f soit une fonction (prémisse gauche).
  - x ait l'horloge du paramètre attendu par f (prémisse droite).
- **f x a alors pour horloge l'horloge de la valeur retournée par f.**
- **Exemple d'application : (λx.x + 3) 5, résultat 8.**

Règle d'inférence when

$$when : \forall \alpha, \forall X, \alpha \rightarrow (X : \alpha) \rightarrow \alpha \ on \ X$$

- **Le premier paramètre de when est sur une horloge quelconque α.**
- **Le second est une condition, représentée par X, ayant elle aussi pour horloge α.**
- **Le résultat (x when c) est sur une nouvelle horloge, l'horloge α restreinte aux instants où la condition X est vraie.**
- **L'horloge α peut elle-même être de la forme β on y.**

Exercice: écrire la règle pour le current

Preuve

- **L'ensemble des règles forme un système de preuves.**
- **On utilise ce système pour prouver qu'une expression complexe est bien synchronisée.**
- **On calcule en même temps son horloge.**
- **Si une expression n'admet pas de preuve dans le système de type, alors elle est mal synchronisée, rejetée par le compilateur.**

Exemple

Programme à inférer

```
node N (c: bool, a: int when c, b: int)
returns (o: int when c)
let
   o =(a + 2) * (b when c);
tel
```


Exemple

```
node prog1 (B1, B2:bool;
            H1:bool when B1;
            H2:bool when B2;
            X1:int when H1;
            X2:int when H2;
            Y:int when (B1 and B2))
returns (S : int when …)
var Z1:int when …; Z2:int when …;
    Z3:int when …;Z4:int when …;
let Z1 = current(X1);
    Z2 = current(X2);
    Z3 = current(Z1) + current(Z2);
    Z4 =(current(Z3) when (B1 and B2)) + Y;
    S = current(Z4);
tel
```

Structure d'un programme réactif

Structure

```
Système ( E , S )
    mémoire M
    M: =M0
    à chaque période faire
      lire(E)
      S= f (M, E)
      M=g (M, E )
      écrire (S)
```

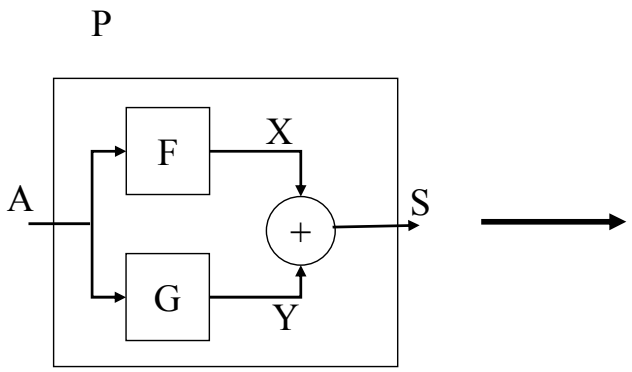
Le compilateur doit :

- Identifier la mémoire M à allouer.
- Calculer les valeurs initiales de M0.
- Fournir le code du corps de la boucle effectué à chaque période.
- En essayant d'optimiser le tout.

Compilation en boucle simple

Un nœud Lustre → une fonction C

Séquentialiser la description




```
int A, X, Y , S;
void F_step () {…}
void G_step () {…}
void P_step () {
  F_step();
  G_step();
  S = X+Y;
}
```

Principes de la compilation en boucle simple

Code Lustre	Code C
Définition de flot	Affectation
Opérateur point à point (+, and)	Opérateur standard
–>	Mémoire initiale
pre	Valeur mémorisée
when	if

Exemples

```
node ajout (i,j :int)
returns (o: int)
let
  o = i+j;
tel
```

```
int i,j,o;
void ajout_step (){
  o=i+j;
 }
```

```
node ajout_1 (i,j :int; c : bool)
returns (o: int when c)
let
  o = (i+j) when c;
tel
```

```
int i,j,o;
bool c;
void ajout_1_step() {
   if ( c ) o=i+j;
              }
```

Exemples

```
node retard (x : int)
returns (y : int)
let
  y = 0 -> pre X
tel
```

```
int x,y; // entrées sorties
 int px; // mémoires non initialisées
bool init = true; // initialisation
void retard_step() {
  if (init) {
        y=0;
        init = false;
        }
   else {
        y=px;
        }
   px = x;
}
```


Exercice

Donner le code généré pour le programme Lustre suivant:

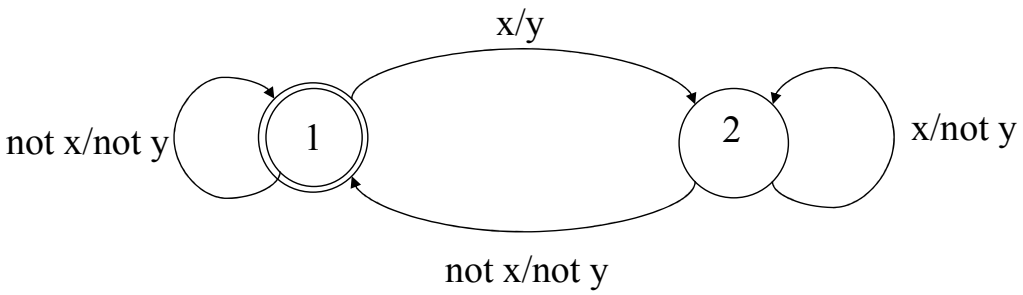
```
node timer (set : bool)
returns (level : bool)
var count : int;
let
 count = if set then 3
         else 0-> (if pre level then pre count -1
                   else pre count);
 level = (count > 0)
tel
```

Compilation en automates

Solution Lustre

– plus puissant, mais compilation plus complexe, code moins lisible.

```
node EDGE (X : bool) returns (Y : bool)
let
   Y = X→(X and not pre(X));
tel ;
```



Etat <==> une configuration de la mémoire

– 1 correspond à px = false
– 2 correspond à px = true

Construction automate réactif

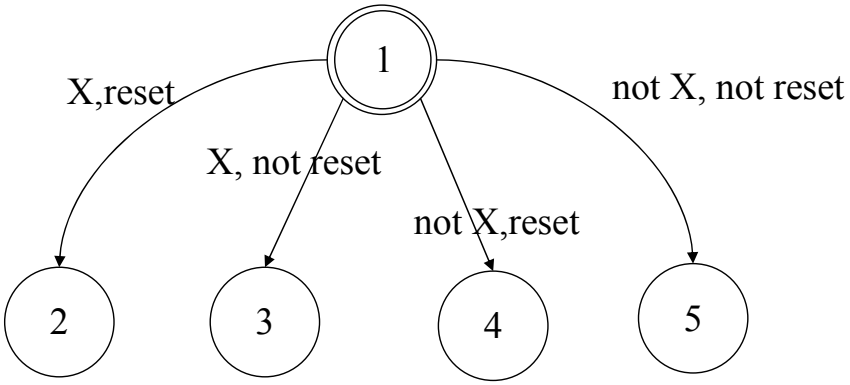
Compteur avec remise à zéro à retard

```
node CptF (X, reset : bool) returns (cpt : int)
var F, R : bool;
let
   cpt = if R then 0
         else if F then pre cpt +1
         else pre cpt;
   F = X→(X and not pre(X));
   R = true -> pre reset;
tel
```

Etats de l'automate: (il y a 3 mémoires)

1. init = true, px = false, preset=false;
2. not init, px, preset
3. not init, px, not preset
4. not init, not px, preset
5. not init, not px, not preset

Construction automate réactif



Et ainsi de suite on calcule les transitions pour les autres états
On minimise l'automate afin d'optimiser

Implantation en C

```
typdef enum {S1, S2, S3, S4, S5} Tstate;
Tstate state = S1;
void CptF_step() {
  switch (state) {
    case S1 | S2 | S4: cpt = 0; break;
    case S3: break;
    case S5: if X cpt++; break;
  }
  if (reset && state == S1) state = S2;
  …

}
```

Exercice

Traiter les nœuds ajout, retard et timer.

Conclusion

Automate

- Compilation plus rapide
- Code potentiellement énorme
- Utile pour la validation

Boucle simple

- Moins rapide
- Taille du code linéaire (solution choisie par les industriels)

Organisation du cours

1. **Présentation de Lustre**
   1. Syntaxe
   2. Sur-échantillonnage et sous-échantillonnage

2. **Quelques points sémantiques**
   1. Sémantique
   2. Compilation
   3. Traduction automates Scade

Références

1. **Alain Girault: cours sur Lustre**
2. **Pascal Raymond: cours sur Lustre**
3. **Julien Forget: cours sur les systèmes temps réel**
4. **Nicolas Halbwachs, Pascal Raymond: A tutorial of Lustre**
5. **Nicolas Halbwachs. Synchronous programming of reactive systems. Kluwer Academic Pub., 1993**
6. **Thomas Bochot, Vérification par model-checking des commandes de vol: applicabilité industrielle et analyse de contre-exemples. Thèse, 2009.**
7. **Christophe Ratel, Définition et réalisation d'un outil de vérification formelle de programmes LUSTRE. Thèse 1992.**

References

1. Frédéric Boniol: course on Lustre
2. Alain Girault: course on Lustre
3. Pascal Raymond: course on Lustre. http://www-verimag.imag.fr/~raymond/edu/lustre.pdf
4. Julien Forget: course on real-time systems
5. Nicolas Halbwachs, Paul Caspi, Pascal Raymond, Daniel Pilaud: The synchronous dataflow programming language Lustre (1991)
6. Nicolas Halbwachs. Synchronous programming of reactive systems. Kluwer Academic Pub., 1993
7. Jérôme Ermont. TP on the Lego robot.
8. Esterel technology: SCADE Update, présentation aérospace vallée, 2010. [Est10]

Plan

1. Partie I - Introduction générale aux systèmes temps réel et aux problématiques de conception
2. Partie II – Langage Lustre: syntaxe, sémantique et preuve
3. **Partie III - SDL**
4. Partie IV – Introduction aux automates temporisés

Plan

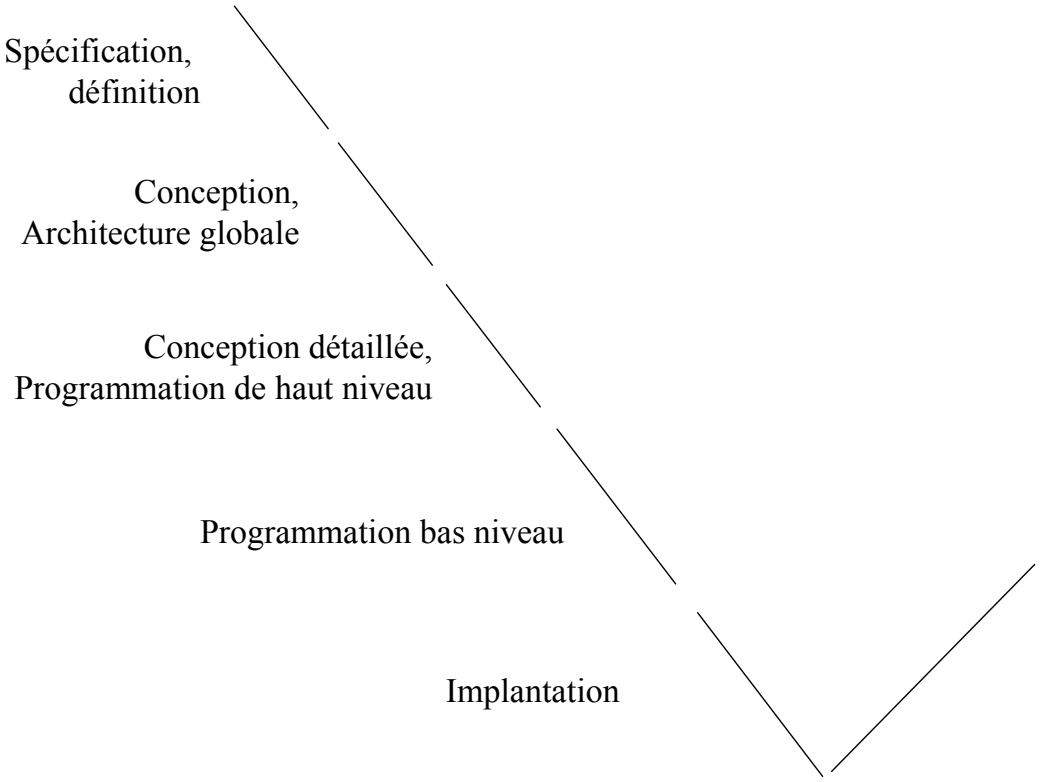
1. **Où peut on utiliser un langage de haut niveau?**

2. **Introduction à SDL**

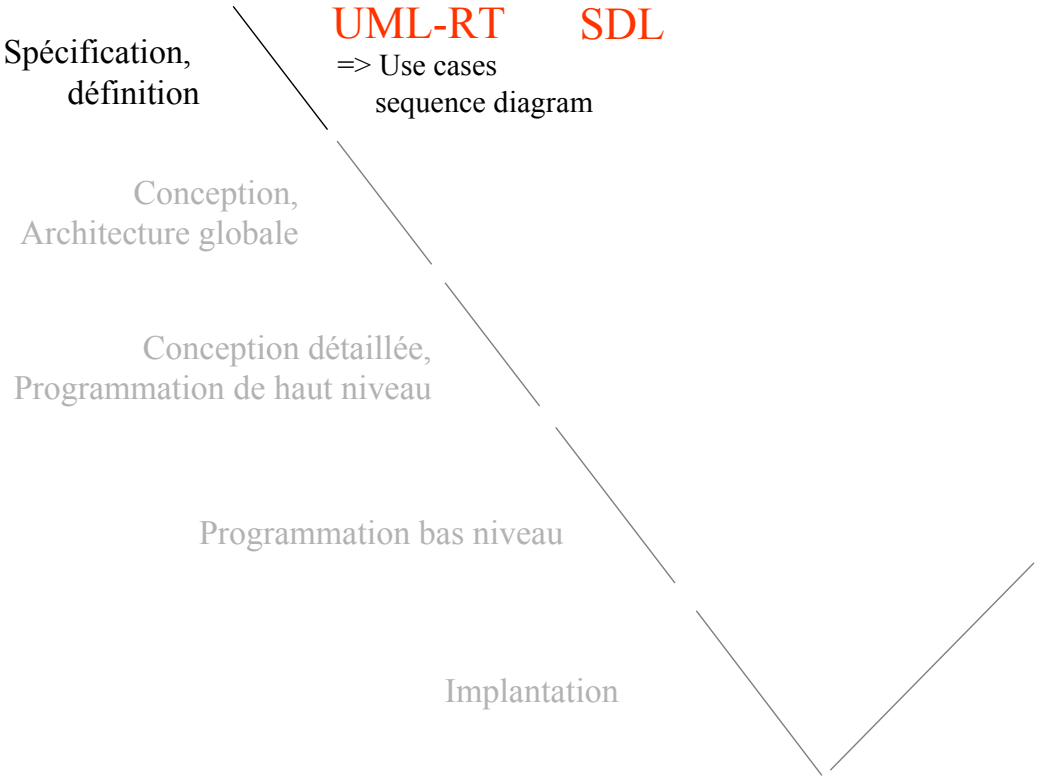
3. **Syntaxe de SDL**

4. **Sémantique de SDL**

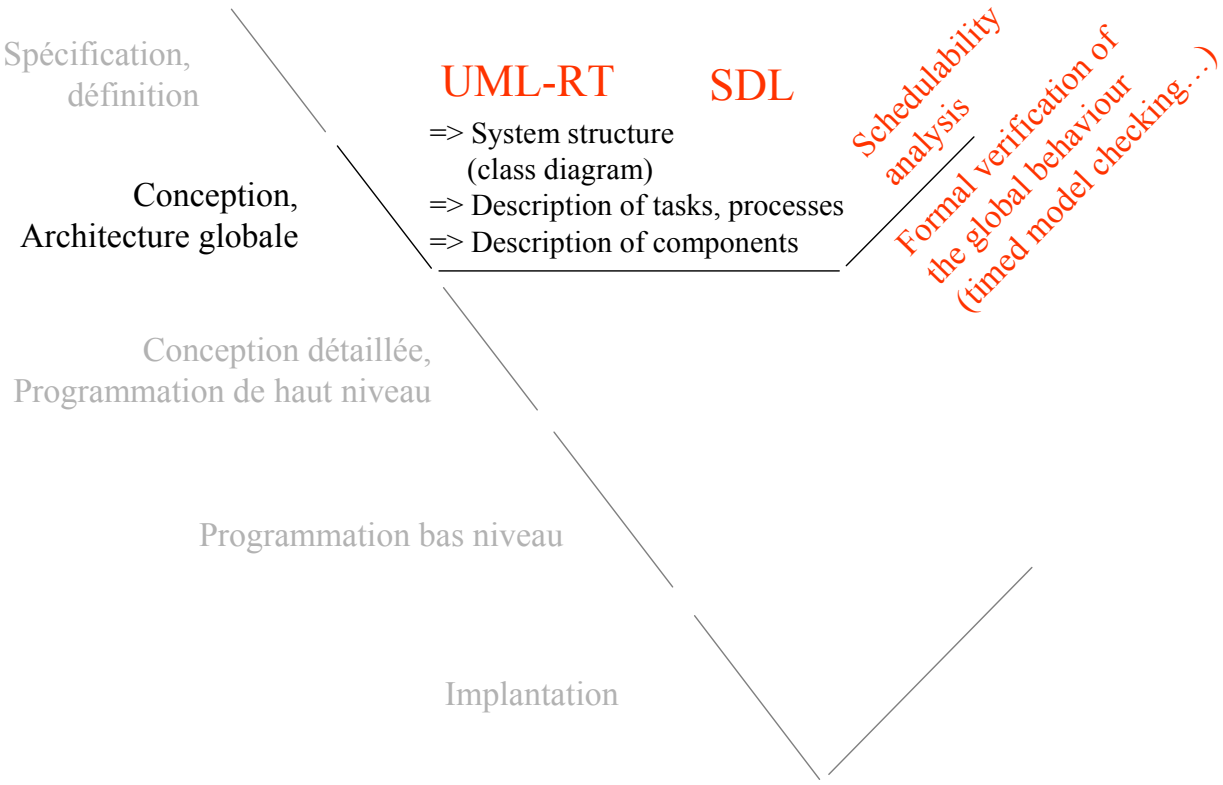
Processus de développement



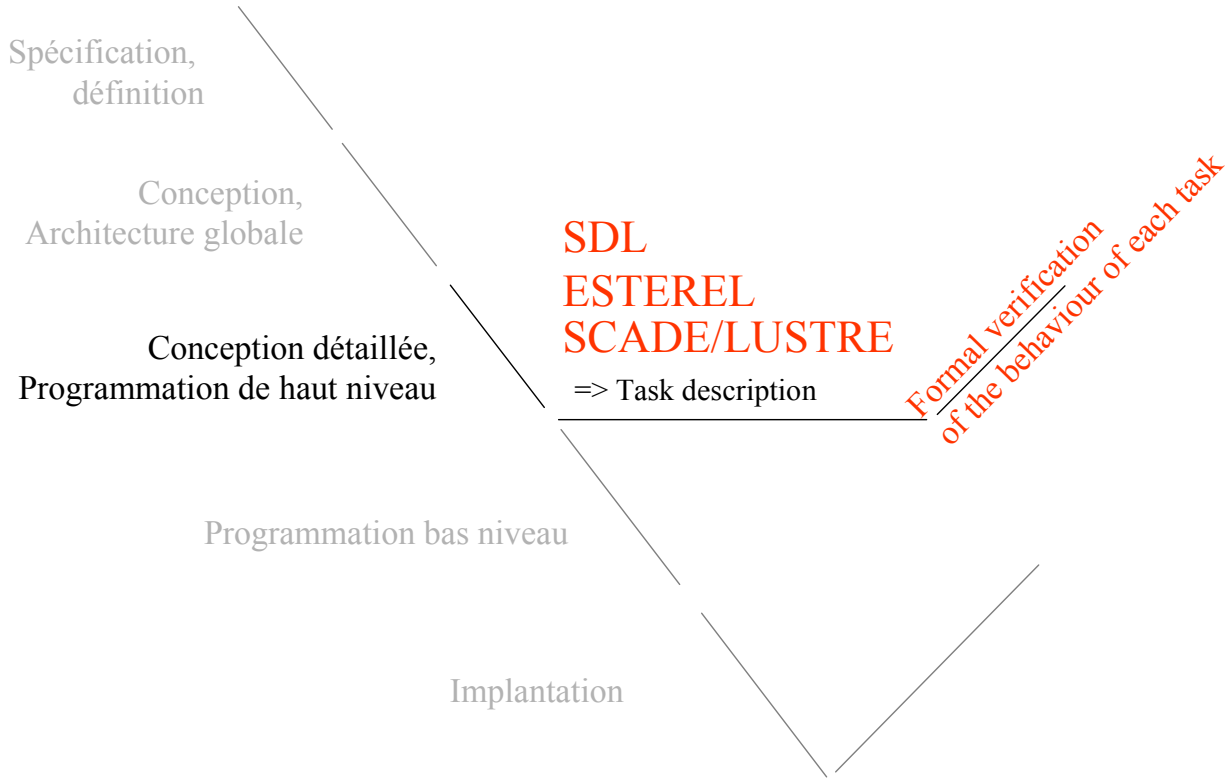
Processus de développement



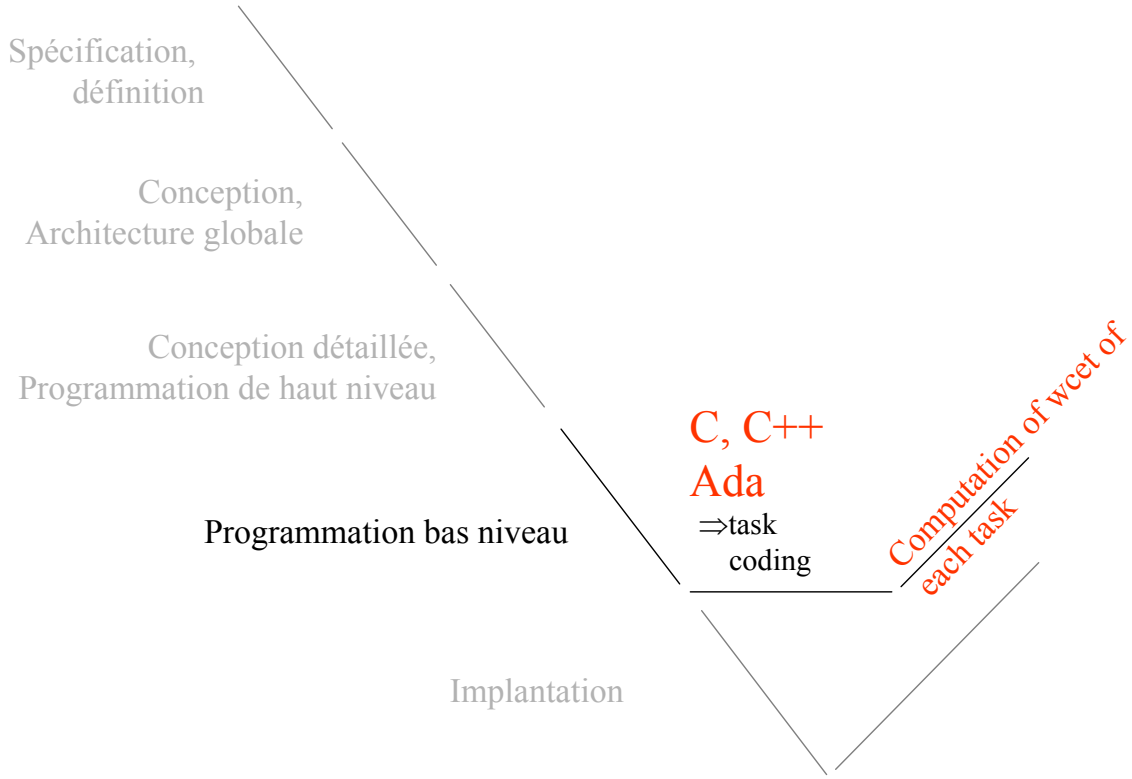
Processus de développement



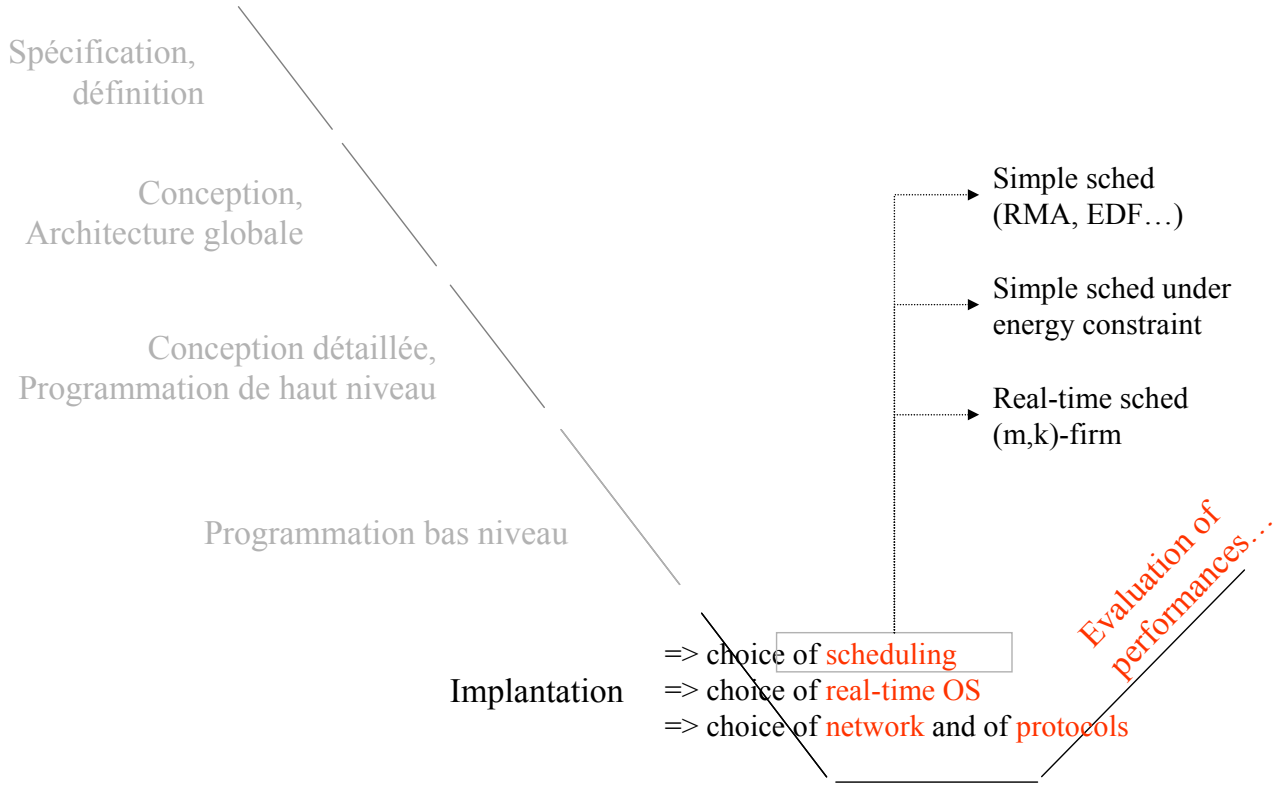
Processus de développement



Processus de développement



Processus de développement



Plan

1. **Où peut on utiliser un langage de haut niveau?**
2. **Introduction à SDL**
3. **Syntaxe et sémantique de SDL**

Introduction

SDL (Specification and Description Language):

- Défini en 1976 par le CCITT, Comité consultatif internationale télégraphique et téléphonique – maintenant ITU-T International Telecommunication Union – recommandation Z.100 – Z.109
- Initialement prévu pour la spécification et le développement de protocoles
- Bien adapté à la description formelle de systèmes réactifs, notamment des systèmes embarqués temps réel distribués
- Langage vivant (mises à jour régulières)

Principales caractéristiques:

- la *spécification* du système est la description du *comportement attendu*
- la *description* du système représente le *comportement réel*
- les spécifications et les descriptions en SDL sont formelles dans le sens où elles peuvent être analysées et interprétées sans ambiguïté (basé sur automates finis communicants étendus avec des types de données abstraits)
- SDL est facile à apprendre et utiliser

  => langage textuel ou graphique
- plusieurs méthodes formelles peuvent être appliquées (model checking, génération de scénario de tests)

Bref historique

1968 premières études

1976 Orange book SDL – langage graphique élémentaire

1980 Yellow book SDL – définition de la sémantique des processus

1984 Red book SDL – types de données, premiers outils

1988 Blue book SDL – concurrence, compositions, outils effectifs

=> SDL-88: langage simple mais sémantique formelle

1992 White book SDL – approche objet

=> SDL-92 langage plus complexe

1996 SDL-96 corrections et ajouts mineurs

2000 Intégration avec UML

=> SDL-2000

Avantages

Spécification: **description du comportement attendu**
Description: **description du comportement réel**
•Possibilité de simuler/exécuter le système partiellement décrit
•Compilateur vérifie la cohérence de l'architecture
•Sémantique formelle: pas d'ambiguïté entre le client et le fournisseur
•Mieux que diagramme de séquence: diagrammes sont incomplets et ne décrivent pas les comportements interdits
•Interface avec UML
•Tests: génération de test automatique et validation

Implantation: **génération de code automatique**
•Indépendance avec l'API
•Génération de code à partir d'un automate peut facilement être fausse

Langage unique tout le long du cycle de développement
A l'ENSEEIHT: RTDS par Pragma-dev

Exemple industriel: ATC (Air Traffic Control)

La France est décomposée en 5 zones chacune contrôlée par un centre ATC. Chaque centre guide les avions, désencombre l'espace aérien et sécurise les trajectoires. Le but est

- de prévenir les collisions entre les aéronefs et le sol ou les véhicules d'une part, et les collisions en vol entre aéronefs d'autre part (autrefois appelés « abordages »);
- de fournir les avis et renseignements utiles à l'exécution sûre et efficace du vol : informations météorologiques, information sur l'état des moyens au sol de navigation, information sur le trafic (quand le service de contrôle n'est pas assuré dans cette zone) ;
- de fournir un service d'alerte pour prévenir les organismes appropriés lorsque les aéronefs ont besoin de l'aide des organismes de secours et de sauvetage, et de prêter à ces organismes le concours nécessaire.

[wikipedia]

ATC embarqué: la liaison de donnée

Les données des calculateurs de bord (position, altitude, vitesse, météo) sont collectées, puis transmises à intervalles réguliers par satellite vers les équipements au sol. Messagerie électronique pour les dialogues entre pilotes et contrôleurs.

- **ADS (Automatic Dependant Surveillance) :** Surveillance automatique de la position réelle de l'avion. Si l'avion dévie de son profil de vol autorisé, il passe dans un mode de surveillance plus serré qui permet de corriger sa position rapidement. Future génération: ADS-B (Broadcast : diffusion), les délais d'envoi sont améliorés.
- **CPDLC (Controller-Pilot Data Link Communications) :** communication entre les pilotes et les contrôleurs par un système de messagerie électronique. Ces dialogues sont codifiés pour des raisons de sécurité (pas d'ambiguïté) : messages préformatés avec passage de paramètres (ex : autorisation de monter ou de descendre à tel ou tel niveau de vol), avec des procédures de bouclage pour s'assurer que l'information a bien été envoyée, reçue et suivie.
- **« surveillance enrichie » ou CAP (Controller Access Parameters).** Les systèmes bord enverront automatiquement des informations de surveillance précises telles que : le cap magnétique, le taux de montée, la vitesse indiquée… Ces informations devenant disponibles pour les contrôleurs aériens, la surveillance des vols devient plus fine et la charge de communication réduite entre pilotes et contrôleurs.

[wikipedia]

ATC embarqué

Application bord: les sous fonctions sont exécutées dans une partition dans un module IMA. Les sous fonctions sont codées par des processus SDL.

456 000 lignes de code C générées à partir de SDL

156 000 lignes de code C manuelles

Plan

1. Où peut on utiliser un langage de haut niveau?

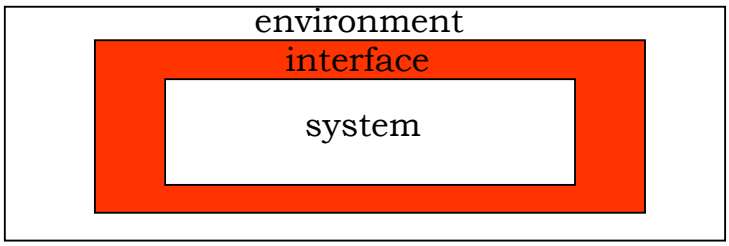
2. Introduction à SDL

3. Syntaxe et sémantique de SDL

Structure of an SDL system

An SDL system is a structured set of processes which execute in parallel and communicate by exchanging messages

An SDL system is a real-time system immersed in its environment through an interface

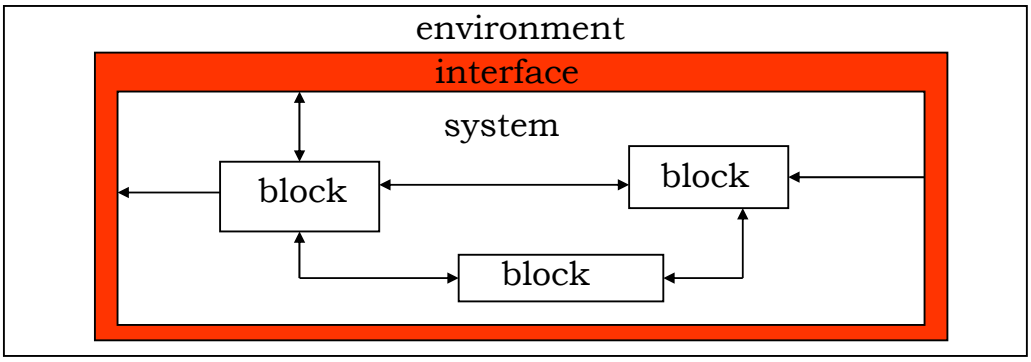


Structure of an SDL system

SDL provides structuring principles which consist in decomposing a system in components that can be developed independently and in no specific order.

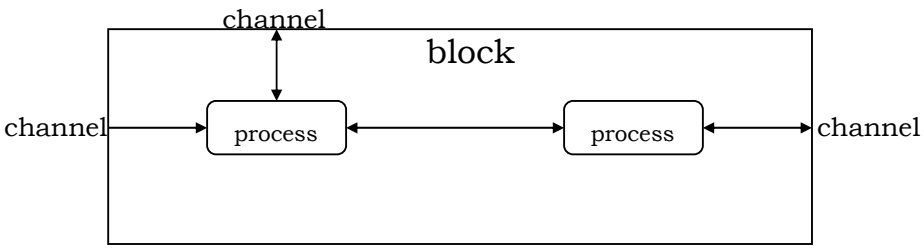
3 levels of structuring:

1. System = block + communication channel
2. Block = processes + signal route
3. Process = an automaton

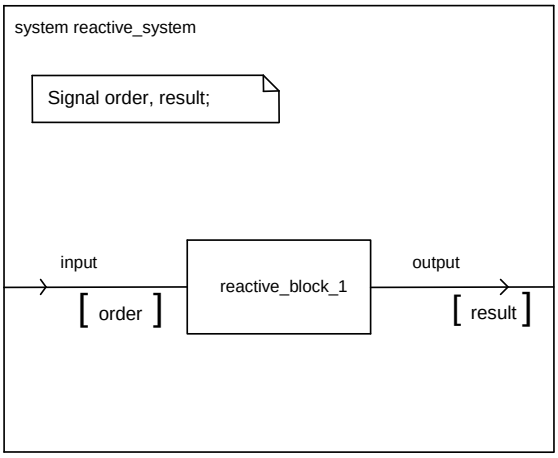


Structure of an SDL system

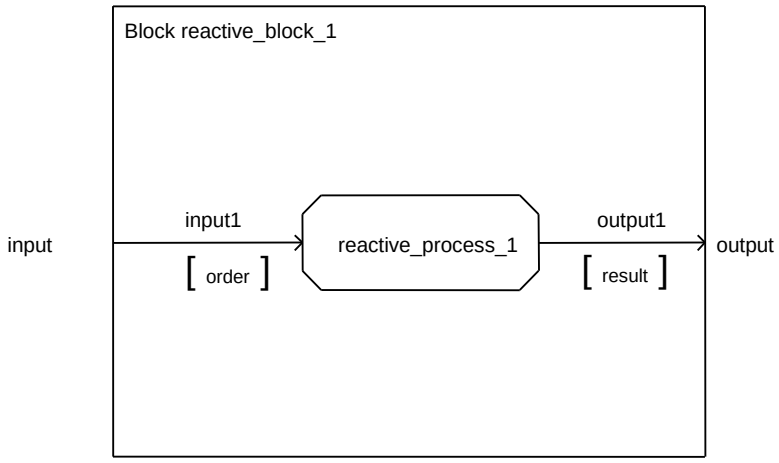
A block can be decomposed into blocks, but a block leaf is necessary a process or a set of processes



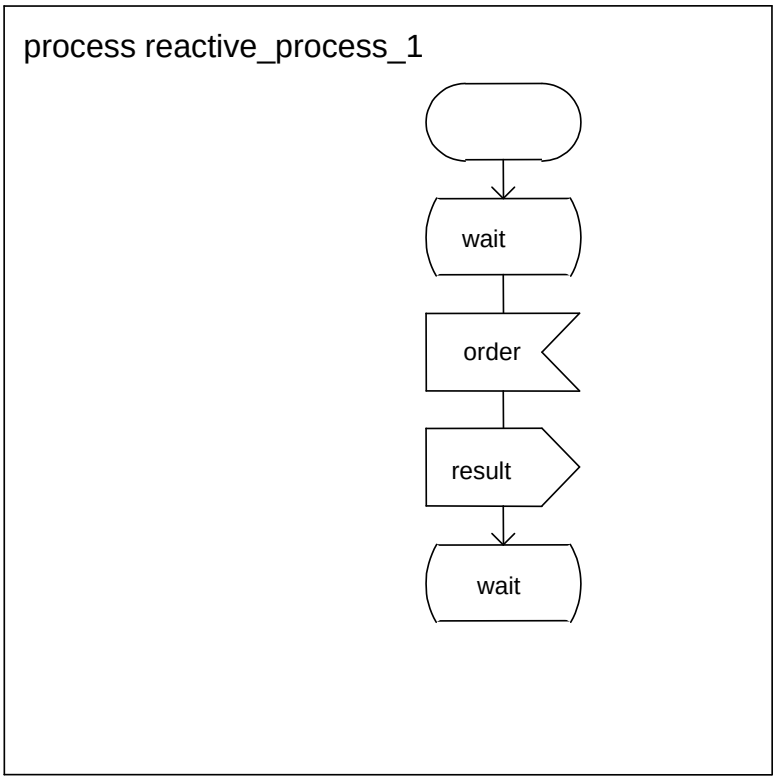
Example: simple reactive system



Example: simple reactive system



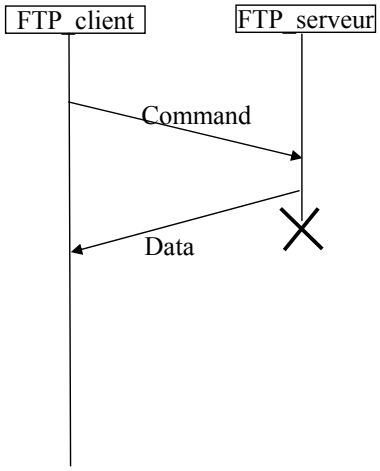
Example: simple reactive system



MSC (Message Sequence Chart)

Les simulateurs de SDL génèrent des MSC. Cela décrit une succession possible des signaux et des mises à jour de variables.

Il faut décrire l'ensemble des processus et l'environnement.



MSC du client/serveur

Lexical rules

Comments:

–between /* …*/

–after keywords COMMENT

–special graphical symbol for the graphical version

a comment

Identifier: [a-z_+#@]*

Channels and signal route are typed

No shared variable

Components are visible by the parents

Declarations

Declarations are made in block text:

At system level, declaration of signal

SIGNAL <ident> (, <ident>)*

Signal Command, data;

A signal is a type, a signal occurrence is a typed message

Elementary types:

–Boolean: true, false

–Character: ‘A’,’1’, …

–Integer: $\mathbb{Z}$

–Natural: $\mathbb{N}$

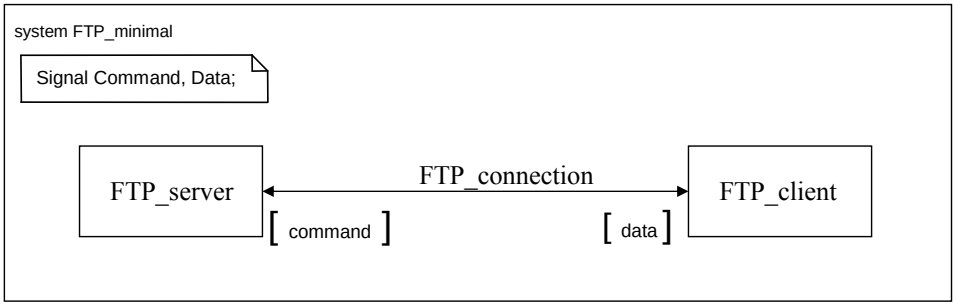
–Real

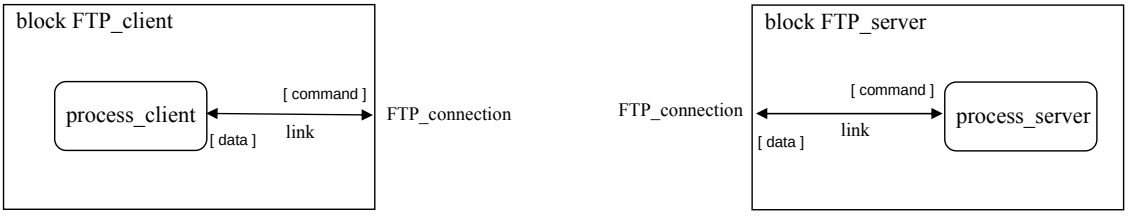
–Charstring

–Pid: identifier of process (2 operators =, /=)

–Time

Example 2

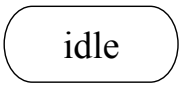


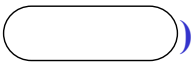


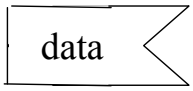
Verification of the coherence of the exchanges

Definition of the behaviour in the process

A process is a finite machine state where the transition depend on the reception of some signal.

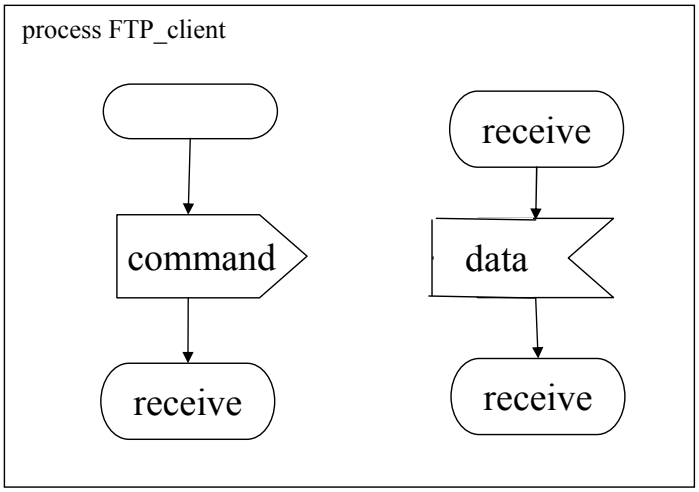
Symbol of a state: 

(initial state  **)**

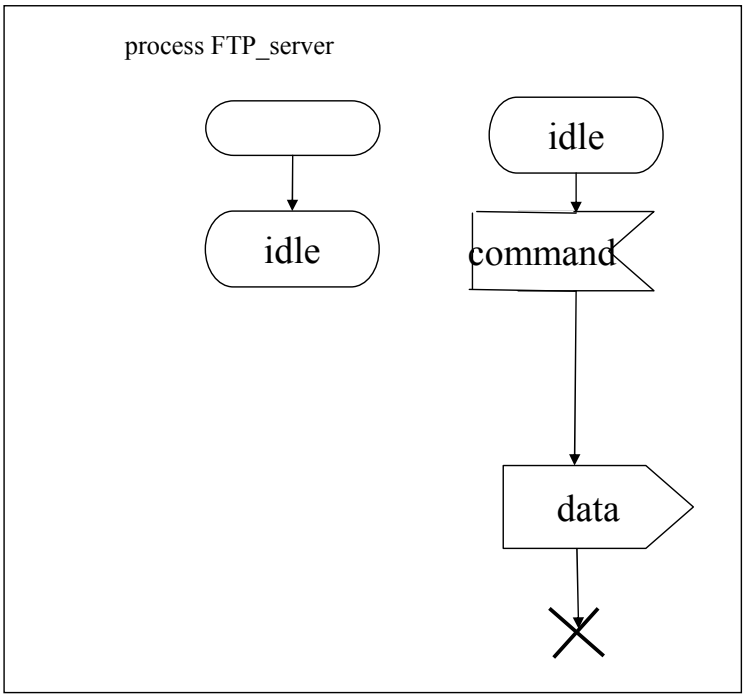
Symbol of reception: 

Symbol of emission: 

Example: process of the client



Example: process of the server



Sémantique d'un processus

Un processus SDL est un système de transition $S=<\Sigma,s_0,T,Sig_{in},Sig_{out},Sv>$:

- Σ est un ensemble fini d'états

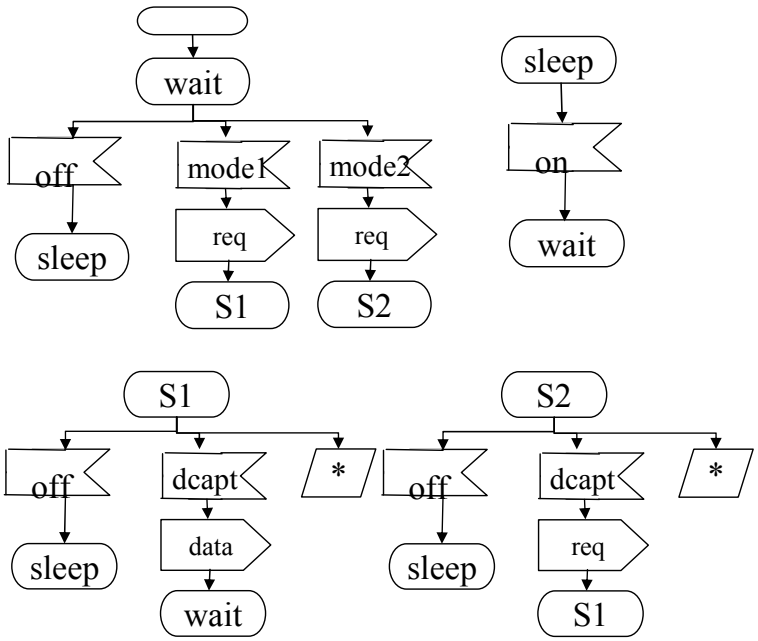
- $s_0 \in \Sigma$ est l'état initial

- Sig_{in} et Sig_{out} sont les signaux d'entrée et de sortie de S

- $T \subseteq \Sigma \times Sig_{in} \times Sig_{out}^{*} \times \Sigma$ ensemble des transitions. Une transition est de la forme (s_1,a,σ,s_2) où s_1 est l'état source, a est l'événement enclenchant la transition, σ la séquence d'événements produits par le processus lors du franchissement de la transition, s_2 l'état d'arrivée

- $Sv : \Sigma \rightarrow 2^{Sig_{in}}$ est la fonction qui associe à chaque état les événements d'entrée sauvegardés.

Exemple



Σ={init, wait, sleep, S1, S2}

Sig_{in} = {off, on, mode1, mode2, dcapt}

Sig_{out} = {req, data}

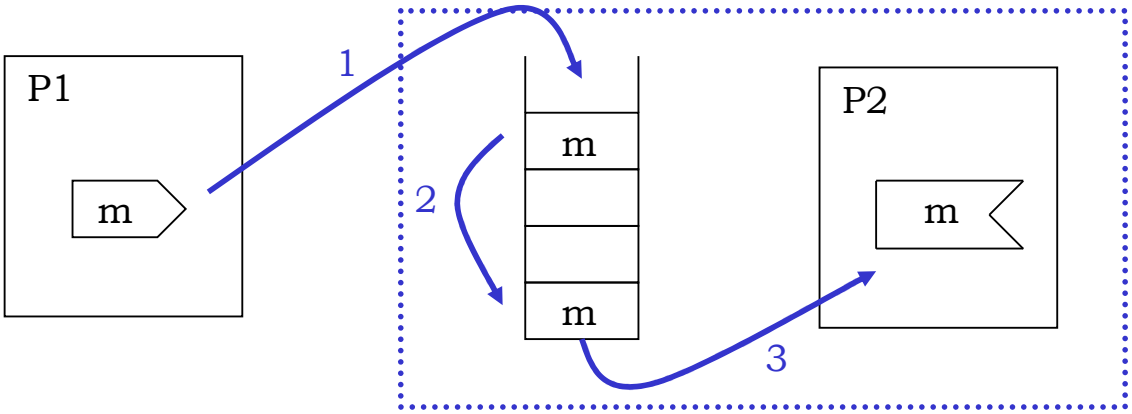
T={ (wait,off,(),sleep); (wait,mode1,(req),S1); (wait,mode2,(req),S2); (sleep,on,(),wait); (S1,off,(),sleep); (S1,dcapt,(data),wait); (S2,off,(),sleep); (S2,dcapt,(req),sleep);

Sv(wait)=Sv(sleep)=();

Sv(S1)=Sv(S2)={on,mode1,mode2}.

Communication semantics

- Communication between processes is asynchronous
- To each process, is associated a unique unbounded FIFO file

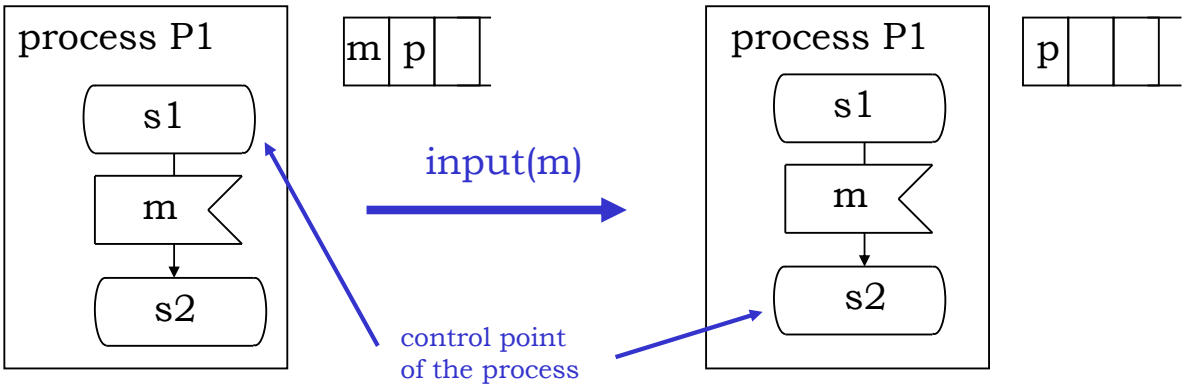


Communication semantics

At the reception of a message:

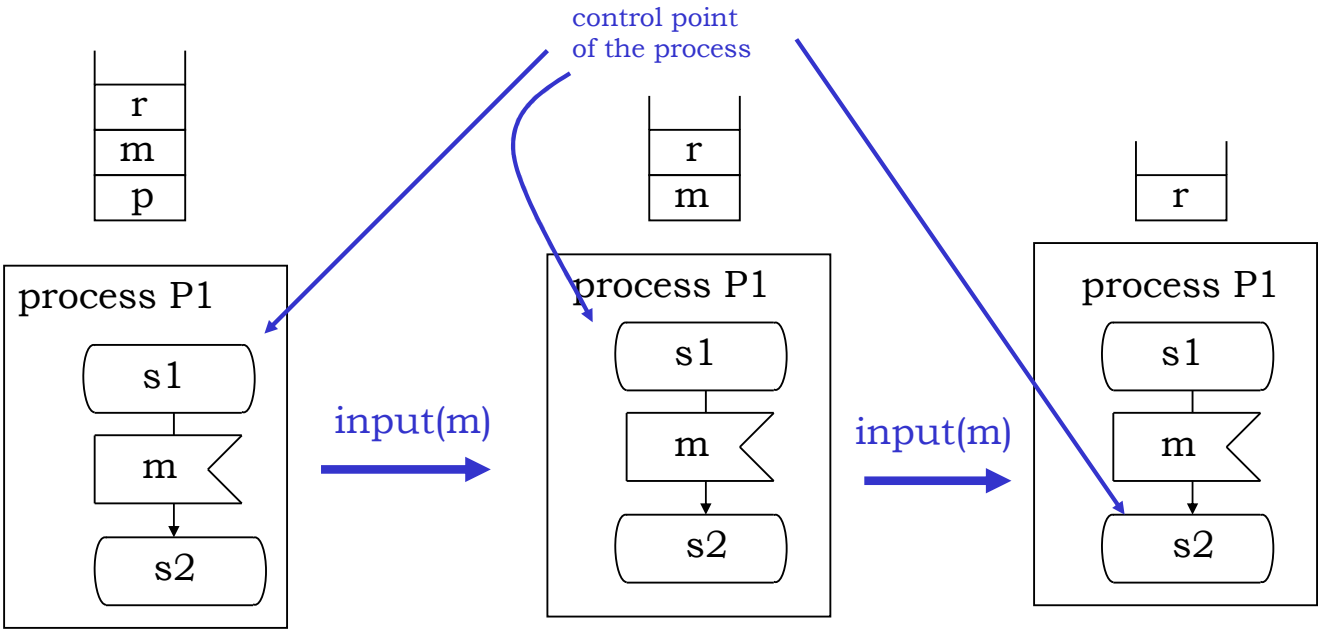
–The message is removed from the file

–Progress of the process



Communication semantics

Unexpected messages are deleted.



Signal handling

The signal received by a process are stored in a FIFO.

In a state, only a subset of signals are awaited. By default, any signal in the head of the FIFO which is not awaited is deleted.

A back-up is possible by the use of the symbol

Variables declaration

In a block text.

DCL <ident> (,<ident>)* <type> (, <ident> (,<ident>)* <type>)*

```
DCL a Character;
DCL b Character, i integer;
DCL c,d Character, n,m Natural ;
```

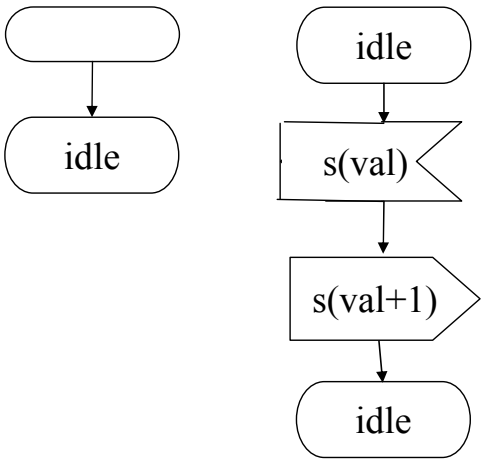

Declarations

Signals can transport some values:

Signal Data(Integer, Boolean);

Signal Ack(Boolean);

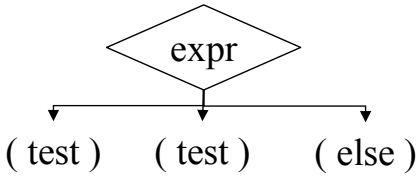
Example: SIGNAL s(Integer);



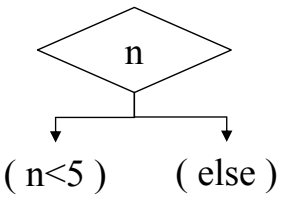

```
DCL val Integer;
```

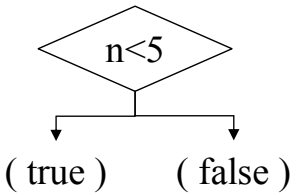
Decision

During a transition (reception of a signal), it is possible to make a decision depending on some expression.



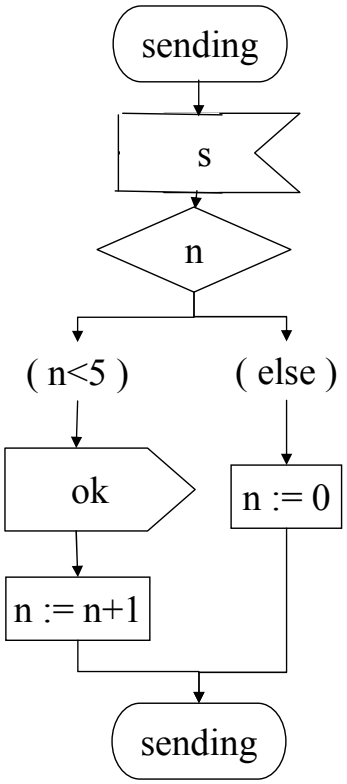
Examples





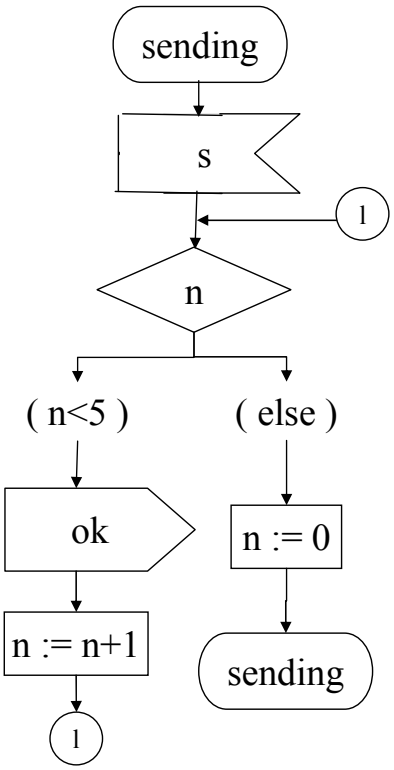
Tasks

After a transition, it is possible to make some computation step task



Loops

At any time after entering in a transition, it possible to put a loop



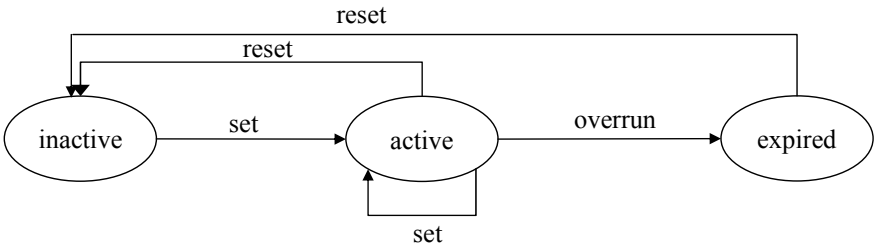
Timers

Variables of type TIMER. Such a variable has 3 states: *inactive*, *active* and *expired*.

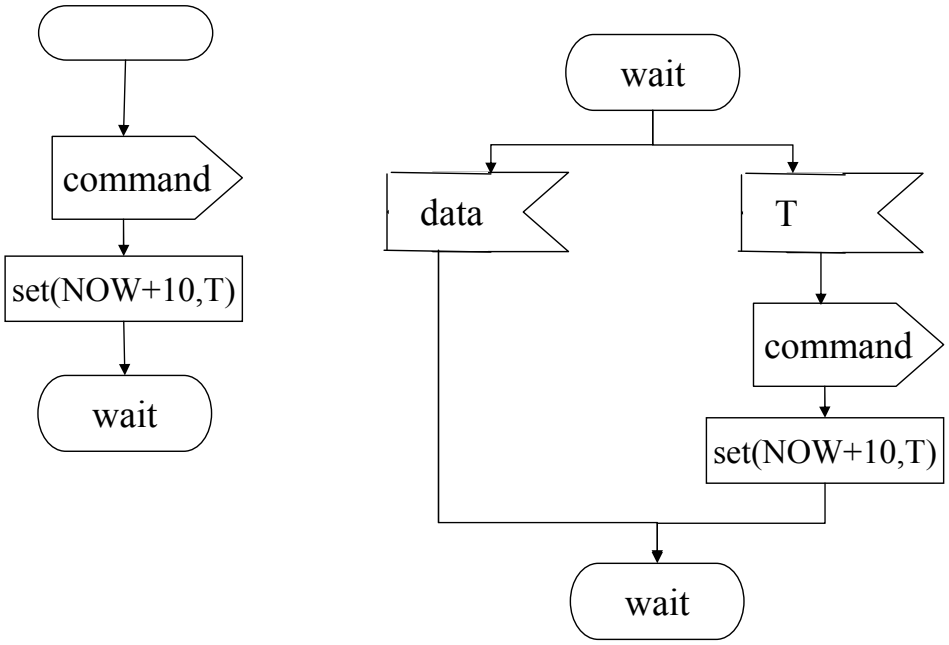
When declared, a timer is inactive. We set a timer by giving it a delay:

SET (<delay>,<timer>)

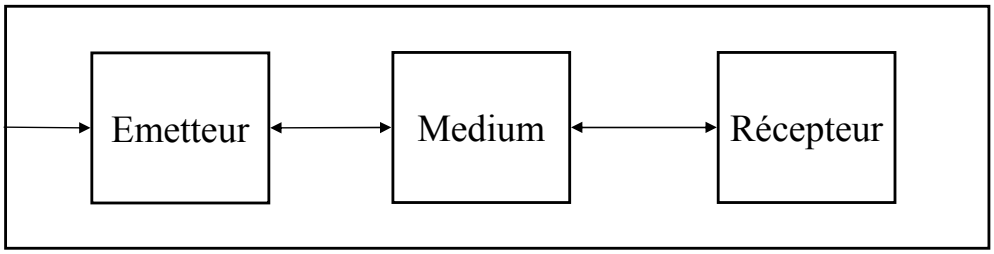
When the delay is overrun, the timer is expired.



Example: process of the client



Exercice



1. **L'utilisateur envoie un signal à l'émetteur qui le transmet au médium lequel est supposé le réémettre au récepteur. Le récepteur renvoie un acquittement dès réception d'un message. Malheureusement le médium perd 1 paquet sur 3 et duplique un acquittement 1 sur 5.**
2. **L'émetteur réémet sa donnée lorsqu'il n'a pas reçu d'acquittement au bout de 10 u.t.**

Références

1. Marc Boyer: course sur SDL.
2. Ahmed Bouadballah: course sur SDL.
3. Jan Ellsberger, Dieter Hogrefe and Amardeo Sarma: SDL formal object-oriented language for communicating systems, Prentice hall, 1997.
4. Laurent Doldi. SDL illustrated – visually design executable models. Number ISBN 2-9516600-0-6. 2001.
5. Specification and description language (SDL). Recommandation Z.100, International Telecommunication union, 1999.
6. Eric Bonnafous, Frédéric Boniol, Philippe Dhaussy et Xavier Dumas. Experience of an efficient and actual MDE process : design and verification of ATC onboard system, 2008, Conference on UML&FORMAL METHODS, Kitakyushu-city, Japan
7. Xavier Dumas. Application des méthodes par ordres partiels à la vérification formelle de systèmes asynchrones clos par un contexte: application à SDL. Thèse 2011.

Plan

1. Partie I - Introduction générale aux systèmes temps réel et aux problématiques de conception
2. Partie II – Langage Lustre: syntaxe, sémantique et preuve
3. Partie III - SDL
4. **Partie IV – Introduction aux automates temporisés**

IV – Introduction aux automates temporisés

1. **Rappels sur les automates**

2. **Définition automates temporisés**

3. **Vérification de propriétés temporelles**

Définition automate

Un automate A est un 4-uplet $(Q,E,s_0,\rightarrow)$ avec:

- Q est un ensemble fini d'états;
- s_0 Є Q est l'état initial;
- E est un alphabet fini d'actions;
- → \subseteq Q x E x Q est une relation de transition.

Exercice

– Modéliser un interrupteur à l'aide d'un automate.

–Modéliser un distributeur automatique de billets (DAB) à l'aide d'un automate. Les spécifications sont les suivantes:

- si l'utilisateur se trompe de code 3 fois de suite, la carte est avalée,
- l'utilisateur ne peut sortir sa carte que s'il a réussi à entrer correctement son code,
- le distributeur ne propose pour unique opération que le retrait de billets: il rend d'abord la carte puis la somme demandée.

Rappels définitions

Définition 2 (Chemin-trace-langage) *Soit $\mathcal{A} = (Q, E, s_0, \rightarrow)$ un automate :*

1. $\sigma = s_0 \rightarrow^{l_0} s_1 \ldots \rightarrow^{l_n} s_{n+1}$ *est un chemin (ou une exécution) fini si* $\forall i, (s_i, l_i, s_{i+1}) \in \rightarrow$.
2. $l_0 \ldots l_n$ *est la trace de (ou mot associé à)* σ.
3. *le langage* $L(\mathcal{A})$ *reconnu par l'automate* $\mathcal{A}$ *est l'ensemble des mots de toutes les exécutions possibles de l'automate.*

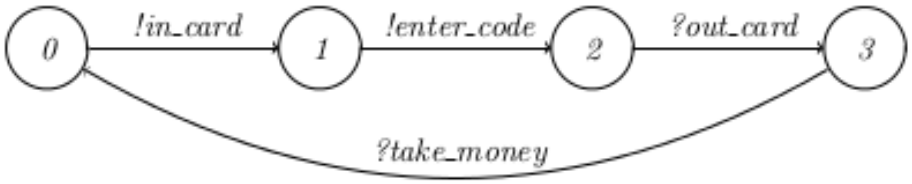
Définition 3 (Produit synchronisé d'automates) *Soient n automates* $\mathcal{A}_i = (Q_i, E_i, s_0^i, \rightarrow_i)$ *(on suppose que* $\epsilon \in E_i$*) et une contrainte de synchronisation* $I \subseteq E_1 \times \ldots E_n$. *Le produit synchronisé* $\|_I \mathcal{A}_i$ *des automates est un automate* $\mathcal{A} = (Q, E, s_0, \rightarrow)$ *défini par :*

1. $Q = Q_1 \times \ldots \times Q_n$,
2. $s_0 = (s_0^1, \ldots, s_0^n)$,
3. $E = E_1 \times \ldots \times E_n$,
4. $\rightarrow \subseteq Q \times E \times Q$ *satisfait :*

$$(q_1, \ldots, q_n) \rightarrow^{(e_1, \ldots, e_n)} (q'_1, \ldots, q'_n) \Longleftrightarrow \begin{cases} (e_1, \ldots, e_n) \in I \\ q_i \rightarrow^{e_i} q'_i \end{cases}$$

Exemple de synchronisation

Reprenons le cas du distributeur de billets. On modélise un comportement idéal de l'utilisateur par l'automate suivant:



Décrire la contrainte de synchronisation entre l'utilisateur et le DAB.

Dessiner le produit synchronisé des deux automates.

IV – Introduction aux automates temporisés

1. Rappels sur les automates

2. Définition automates temporisés

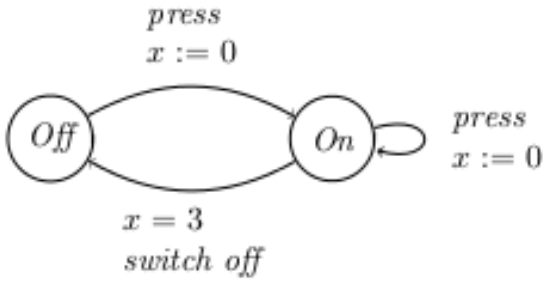
3. Vérification de propriétés temporelles

Qu'est-ce qu'un automate temporisé?

Les automates temporisés ont été introduits au début des années 90 par Rajeev Alur et David Dill.

Le formalisme des automates temporisés peut servir à modéliser (ou spécifier) un système temps réel. On peut alors vérifier des propriétés sur ce modèle ou générer du code pour l'implanter.

Dans la pratique, un automate temporisé est un automate étendu avec des *horloges* modélisant des contraintes et comportements temporels.



Une minuterie

Qu'est-ce qu'un automate temporisé?

On rajoute des variables particulières sur les automates, appelées horloges. Initialement toutes les horloges commencent à 0 et augmentent au même rythme.

Sur une transition, on peut

- tester la valeur d'une (ou de plusieurs) horloge(s);
- remettre une (ou plusieurs) horloge(s) à zéro.

Exercice:

Modifier le DAB pour ajouter le comportement suivant: le code doit être entré en moins de 6 secondes.

Valuations

Notations Soit X un ensemble d'horloges. Une valuation de X est une fonction $v : X \to \mathbb{R}_+$. L'ensemble des valuations est noté $\mathbb{R}_+^X$. Si $v \in \mathbb{R}_+^X$ est une valuation et $t \in \mathbb{R}_+$, on écrit $v + t$ pour la valuation qui associe à toute horloge x la valeur $v(x) + t$. Soit $Y \subseteq X$, $v[Y \mapsto 0]$ dénote la valuation qui associe 0 pour valeurs aux horloges de Y et $v(x)$ aux autres horloges $x \in X \setminus Y$. On écrit 0_X pour la valuation qui associe 0 à toutes les horloges de X.

L'ensemble des contraintes d'horloges sur X, noté $\mathcal{C}(X)$, est l'ensemble des contraintes définies par la grammaire suivante :

$$\mathcal{C}(X) \ni g ::= x \sim r | x - y \sim r | g \wedge g$$

avec $x, y \in X$, $\sim \in \{<, \leq, =, \geq, >\}$ et $r \in \mathbb{N}$. Une contrainte de la forme $x \sim r$ ou $x - y \sim r$ est appelée une *contrainte atomique*. Le deuxième type est appelé une *contrainte diagonale*.

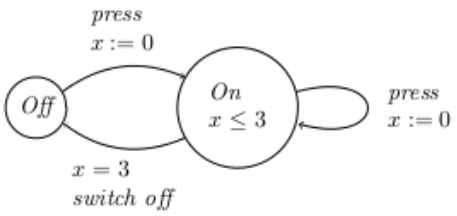
Exemple 5 *La valuation v sur $\{x, y\}$ telle que $v(x) = 2.3$ et $v(y) = 1$ satisfait la contrainte $(x \leq 4) \wedge (x - y) \geq 0$.*

Définition formelle

Définition 4 (Automate temporisé) *Un automate temporisé $\mathcal{A}$ est un 5-uplet $(L, E, l_0, X, \rightarrow)$ avec :*

1. *L est un ensemble fini localités ;*
2. *l_0 est la localité initiale ;*
3. *E est un alphabet fini d'actions ;*
4. *X est un ensemble fini d'horloges (à valeurs dans $\mathbb{R}^+$) ;*
5. *$\rightarrow \subseteq L \times \mathcal{C}(X) \times E \times 2^X \times L$ est un ensemble fini de transitions. Une transition est de la forme $e = \langle l, \gamma, e, R, l' \rangle \in \rightarrow$ avec γ est la garde de la transition, e est l'action et R est l'ensemble des horloges remises à zéro.*

On peut également ajouter un invariant Inv à chaque localité de la forme $Inv(l) = \bigwedge c \leq r$ où c est une horloge et $r \in \mathbb{N}$.



Une minuterie avec invariant

Sémantique

Pour donner la sémantique d'un automate temporisé, il faut interpréter les contraintes d'horloges. On interprète les contraintes d'horloge inductivement :

$$\begin{cases} x \sim r(v) = true & si\ v(x) \sim r \\ x - y \sim r(v) = true & si\ v(x) - v(y) \sim r \\ g_1 \wedge g_2(v) = true & si\ g_1(v) \wedge g_2(v) \end{cases}$$

On définit alors $[\![g]\!] = \{v \in \mathbb{R}_+^X \,|\, g(v) = true\}$.

Définition 5 (Sémantique d'un automate temporisé) *La sémantique d'un automate temporisé $\mathcal{A} = (L, E, l_0, X, \rightarrow)$ est un système de transition (temporisé) $\mathcal{T}_\mathcal{A} = (Q, A, s_0, \rightarrow)$ avec :*

1. *$Q = L \times \mathbb{R}_+^X$ l'ensemble des états ou configurations,*
2. *$s_0 = (l_0, 0_X)$ l'état initial,*
3. *$A = E \cup \mathbb{R}_+$,*
4. *$\rightarrow \subseteq Q \times A \times Q$ est défini par deux types de transition :*
   (a) *transition discrète :*
   $$(l, v) \rightarrow^e (l', v')\ ssi\ \exists (l, \gamma, e, R, l') \in \rightarrow\ \ t.q.\ \gamma(v) = true \wedge v' = v[R \mapsto 0]\ (\wedge\ Inv(l')(v') = true)$$
   (b) *transition temporelle :*
   $$(l, v) \rightarrow^t (l', v')\ ssi\ l = l' \wedge v' = v + t\ (\wedge \forall t' \leq t,\ \ Inv(l)(v + t') = true)$$

Exercice 3 *Donner la sémantique de l'automate de la minuterie de l'exemple 4.*

Chemins

Définition 6 (Chemin-mot-langage temporisés) *Soit $\mathcal{A} = (L, E, l_0, X, \rightarrow)$ un automate temporisé et $L_F \subseteq L$ un ensemble d'états finaux :*

1. *Une exécution (ou chemin) de $\mathcal{A}$ est une suite finie $\sigma = s_0 \rightarrow^{t_1} s'_0 \rightarrow^{e_1} s_1 \ldots \rightarrow^{e_n} s_n$ partant d'une configuration initiale et où les délais alternent strictement avec les actions. L'événement e_i se produit à la date $\Sigma_{j \leq i} t_j$. On écrit généralement une exécution sous la forme canonique : $s_0 \rightarrow^{t_1} s'_0 \rightarrow^{e_1} s_1 \equiv s_0 \rightarrow^{(t_1, e_1)} s_1$. On dit que l'exécution est acceptée par l'automate si $s_n \in L_F$.*
2. *Le mot temporisé $w = (e_1, t_1)(e_2, t_2) \ldots (e_n, t_n)$ correspond à l'observation de la suite des événements du chemin σ. C'est un mot sur l'alphabet $E \times \mathbb{R}_+$.*
3. *La durée de σ est donc $\Sigma_{j \leq n} t_j$. La trace non temporisée associée à l'exécution σ est $e_1 \ldots e_n$.*
4. *Le langage temporisé $L(\mathcal{A})$ accepté par $\mathcal{A}$ est l'ensemble des mots temporisés associés à toutes les exécutions acceptées par l'automate.*

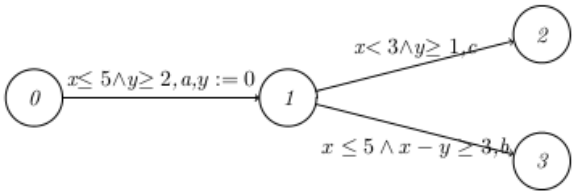
Exemple 7 *Reprenons l'exemple de la minuterie. Initialement le système est dans l'état $(Off, 0)$. Une exécution dans ce modèle est décrite par l'évolution des états et celle des horloges. Voici une exécution possible :*

$$(Off, 0) \rightarrow^{6.9} (Off, 6.9) \rightarrow^{press} (On, 0) \rightarrow^{1.8} (On, 1.8) \rightarrow^{press} (On, 0) \rightarrow^{3} (On, 3) \rightarrow^{switch\ off} (Off, 3)$$

Donner une exécution dans le modèle sans invariant qui n'existe pas dans le modèle avec invariant.

Produit synchronisé

Exercice 4 *On considère l'automate suivant :*



Est-ce que les mots temporisés $w = (a, 2.1)(c, 1.5)$ et $w' = (a, 4.7)(b, 0.2)$ appartiennent à $L(\mathcal{A})$ avec $L_F = \{2, 3\}$?

On peut modéliser des réseaux d'automates temporisés en définissant la notion de produit.

Définition 7 (Produit synchronisé d'automates temporisés) *Soient n automates temporisés $\mathcal{A}_i = (L_i, E_i, l_0^i, X_i, \rightarrow_i)$ (on suppose implicitement qu'on peut faire une transition ϵ sur chaque état) et une contrainte de synchronisation $I \subseteq E_1 \times \ldots E_n$. Le produit synchronisé $\|_I \mathcal{A}_i$ des automates est un automate temporisé $\mathcal{A} = (L, E, l_0, X, \rightarrow)$ défini par :*

1. $L = L_1 \times \ldots \times L_n$,
2. $l_0 = (l_0^1, \ldots, l_0^n)$,
3. $E = E_1 \times \ldots \times E_n$,
4. $X = \cup X_i$,
5. $$(q_1, \ldots, q_n) \rightarrow^{\gamma_1 \wedge \ldots \wedge \gamma_n; (e_1, \ldots, e_n); R_1 \cup \ldots \cup R_n} (q'_1, \ldots, q'_n) \Longleftrightarrow \begin{cases} (e_1, \ldots, e_n) \in I \\ q_i \rightarrow^{\gamma_i; e_i; R_i} q'_i \end{cases}$$

IV – Introduction aux automates temporisés

1. Rappels sur les automates

2. Définition automates temporisés

3. Vérification de propriétés temporelles