Conception de Systèmes Embarqués Temps Réel

Dumitru Potop-Butucaru

dumitru.potop@inria.fr

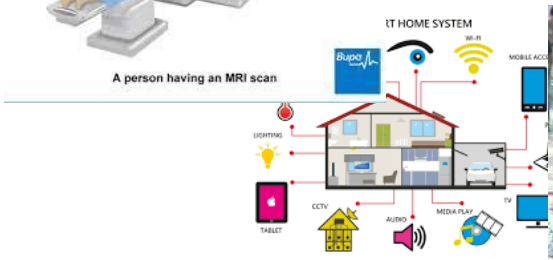
cours EPITA, 2025

Période de changement profond

- Développeur logiciel
  - Réduction de 75% des offres d'emploi par rapport au pic https://fred.stlouisfed.org/graph/?g=1DEP0
  - Réduction de 27% des postes https://fortune.com/2025/03/17/computer-programming-jobs-lowest-1980-ai/
- Le focus se déplace vers le **niveau système** et vers la **specification**
  - **Langages de spécification**, besoin de compilateurs
  - Besoin de **comprendre comment les choix de la specifications impactent l'implantation**

Contenu de ce cours

Système embarqué

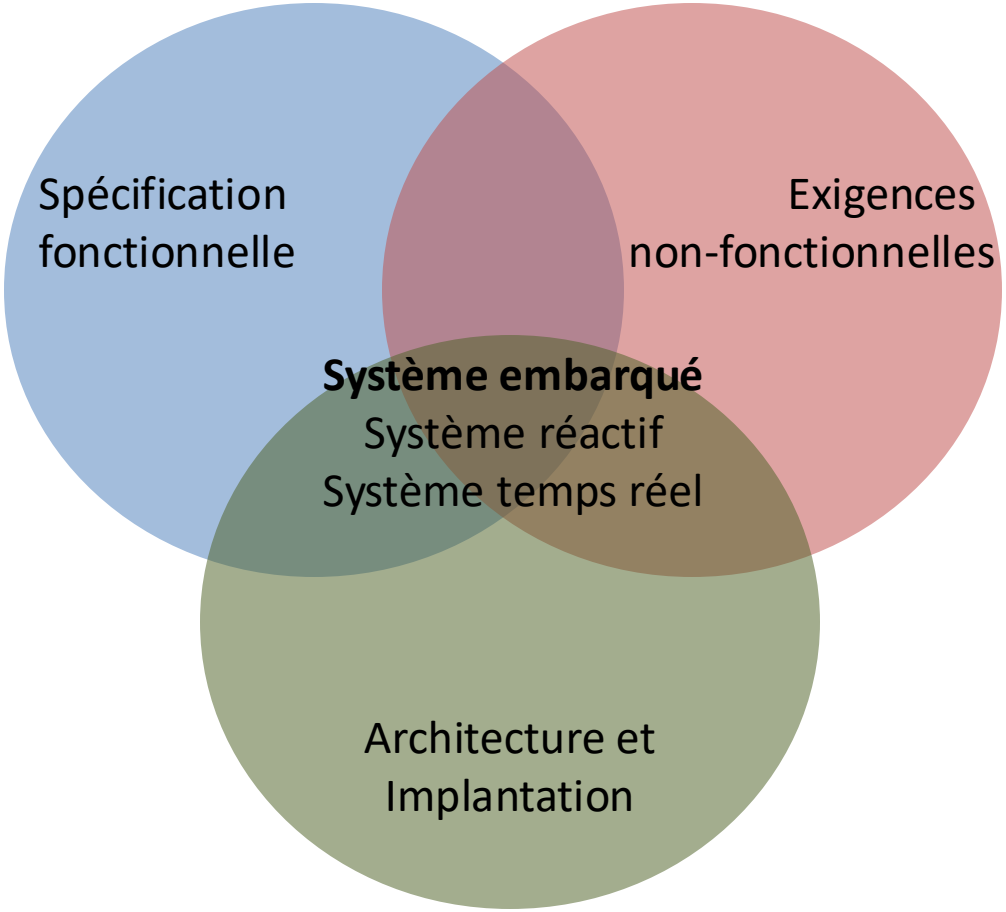


Contenu de ce cours

Système embarqué
Système réactif
Système temps réel

Contenu de ce cours

Langages dédiés
Concepts fondam.

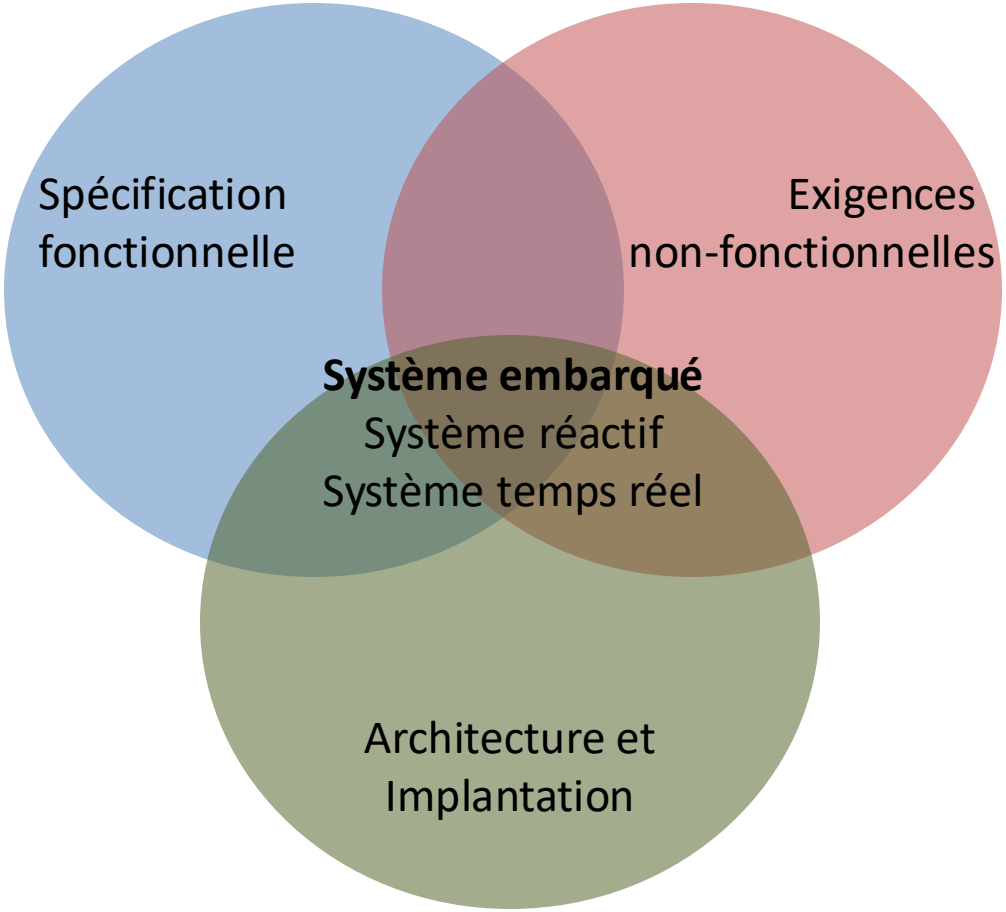


Temps réel
Mémoire
Partitionnement…

Multi-cœurs (organisation, configuration, programmation)
Ordonnancement (en ligne, hors ligne)
Génération de code

Contenu de ce cours

Langages dédiés
Concepts fondam.
Concurrence =
parallélisme
potentiel

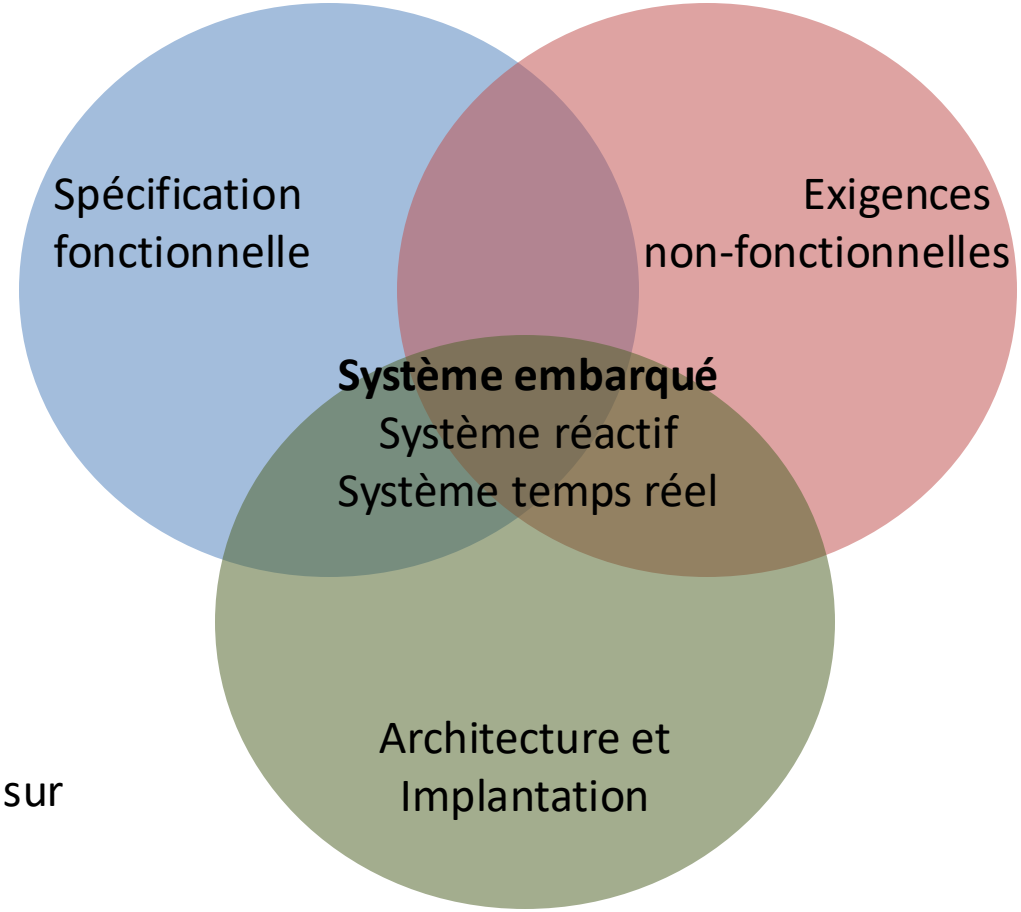


Temps réel
Mémoire
Partitionnement…

Multi-cœurs (organisation, configuration, programmation, parallélisme)
Ordonnancement (en ligne, hors ligne)
Génération de code

Contenu de ce cours

Langages dédiés
Concepts fondam.
Concurrence =
parallélisme
potentiel



Temps réel
Mémoire
Partitionnement…

Approche "hands-on"
- Spécification
- Compilation
- Exécution sur PC et sur SBC Raspberry Pi

Multi-cœurs (organisation, configuration, programmation, parallélisme)
Ordonnancement (en ligne, hors ligne)
Génération de code

Contenu de ce cours – le bas niveau

- C'est un cours intégratif !
  - Connaissances utilisées systématiquement :
    - Programmation en C
    - Compilation de programmes C
      - Compilation séparée (plusieurs fichiers, chacun compilé séparément, avec édition de lien par la suite)
      - Manipulation de répertoires "include" et de répertoires de bibliothèques
      - Makefiles ou scripts sh/bash
    - Utilisation de traçage pour le débogage (pour éviter l'utilisation de sondes JTAG sur cible embarquée)
- Outils open-source
  - Heptagon, gcc (compilateurs), run-time ARM
  - Situation proche d'un environnement de production, où les outils très stables (e.g. gcc) et les outils moins stables se côtoient
    - Vous êtes de futurs ingénieurs !

Points particuliers

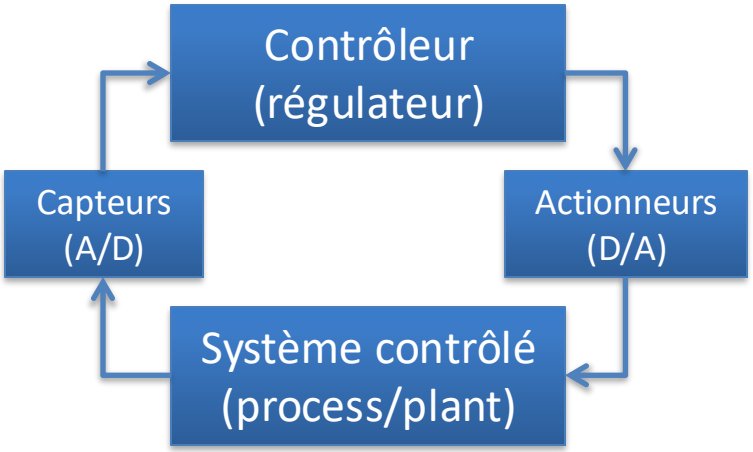
- Tâches temps réel exprimées en **flot de données**
  - Je vous incite fortement à bien vous approprier ce paradigme de programmation
    - Domaines d'utilisation:
      - Embarqué temps réel critique (Lustre/Scade/Heptagon…)
      - Traitement de signal (Simulink, SDF…)
      - Machine Learning – Tensorflow/Jax/Keras, PyTorch, ONNX…
        - » Les aspects avancés traités dans ce cours (état, multi-périodes…) seront de plus en plus utilisés dans le futur
        - » L'embarquement de l'IA est un problème pas bien résolu
- Programmation **bas-niveau et système**
  - Le besoin existera toujours, même si les plates-formes changent
  - Exemple : comment exécuter une application (e.g. ML) sur une carte embarquée

Evaluation

- Contrôle continu (sur le TP)
  - Travail et rendu
    - individuels, les 4 premiers TP
    - par **binôme** ensuite
  - Évaluation par problème résolu
    - N'hésitez pas à m'appeler dès qu'un problème est résolu (ou dès que vous êtes bloqués)
    - Le rendu est attendu, sauf exception notée, le soir après la séance de TP *suivant* celle où le problème a été donné
      - Décote importante par la suite
      - Si vous n'arrivez pas à terminer un problème avant l'échéance, envoyez-moi toujours le code pour l'évaluation (par mél)
      - Rendu: par mél, à l'adresse dumitru.potop@inria.fr
        - » Une archive .tar.gz/.tgz/.zip par objectif
        - » L'archive doit contenir un unique répertoire contenant tout le code et le Makefile
        - » Nom attendu de l'archive et du répertoire: tp<N>_obj<M>_<NOM ETUDIANT>. Exemple: tp1_obj1_potop.tar.gz/tp1_obj1_potop

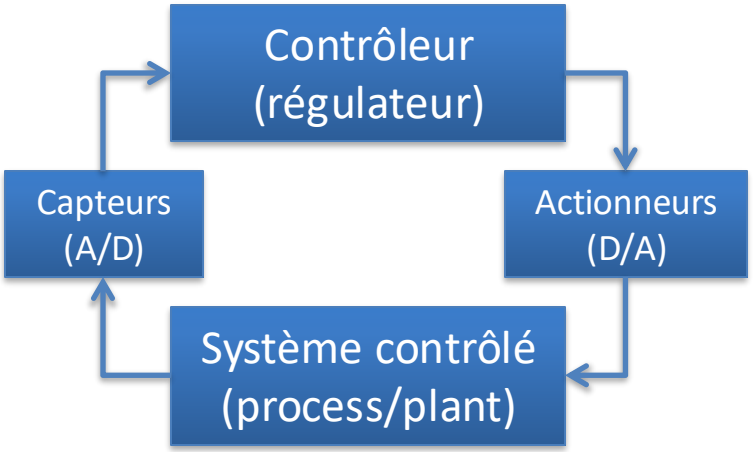
Système de **contrôle embarqué**

- Système informatique qui contrôle un système « physique » englobant
  - Système « physique »:
    - Avion, train, routeur réseau…
  - Forme générale
    - Contrôle en boucle fermée



Système **cyber-physique** vs. Système de **contrôle embarqué**

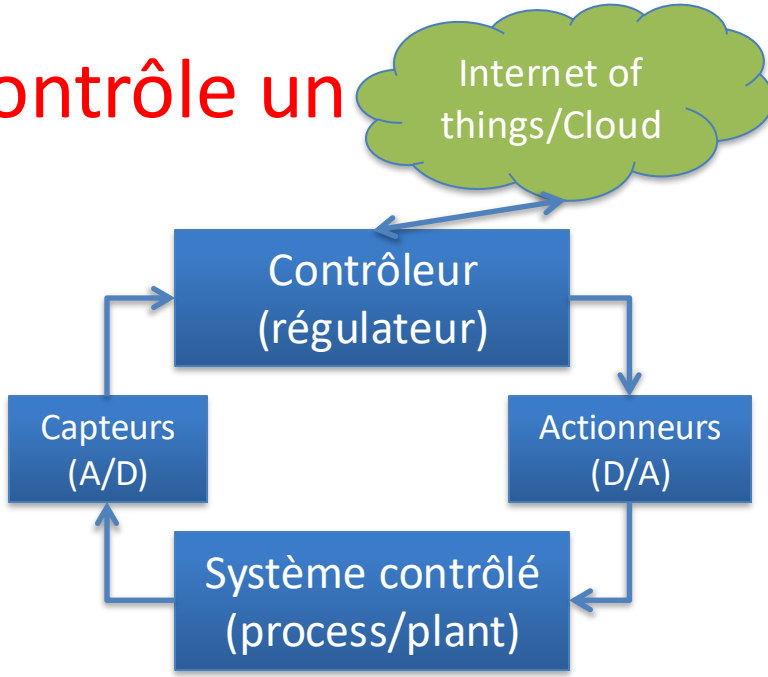
- Système informatique qui contrôle un système « physique » englobant
  - Cyber-physique – accent mis sur l'interaction entre environnement physique et système informatique
  - Contrôle embarqué – accent mis sur le contrôleur informatique



Système **cyber-physique** vs. Système de **contrôle embarqué**

Edge computing

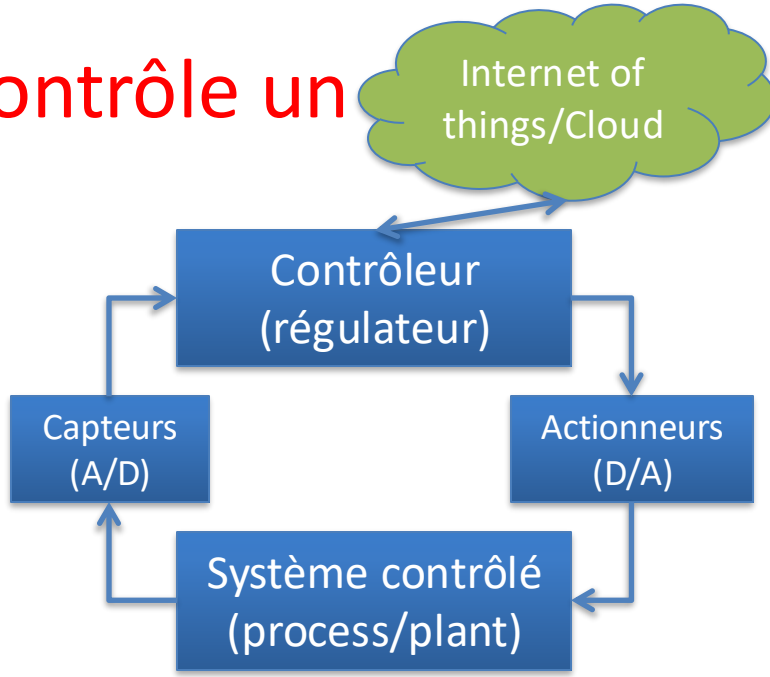
- Système informatique qui contrôle un « physique » englobant
  - Cyber-physique – accent mis sur l'interaction entre environnement physique et système informatique
  - Contrôle embarqué – accent mis sur le contrôleur informatique



Système **cyber-physique** vs. Système de **contrôle embarqué**

Edge computing

- Système informatique qui contrôle un « physique » englobant
  - Cyber-physique – accent mis sur l'interaction entre environnement physique et système informatique
  - Contrôle embarqué – accent mis sur le contrôleur informatique

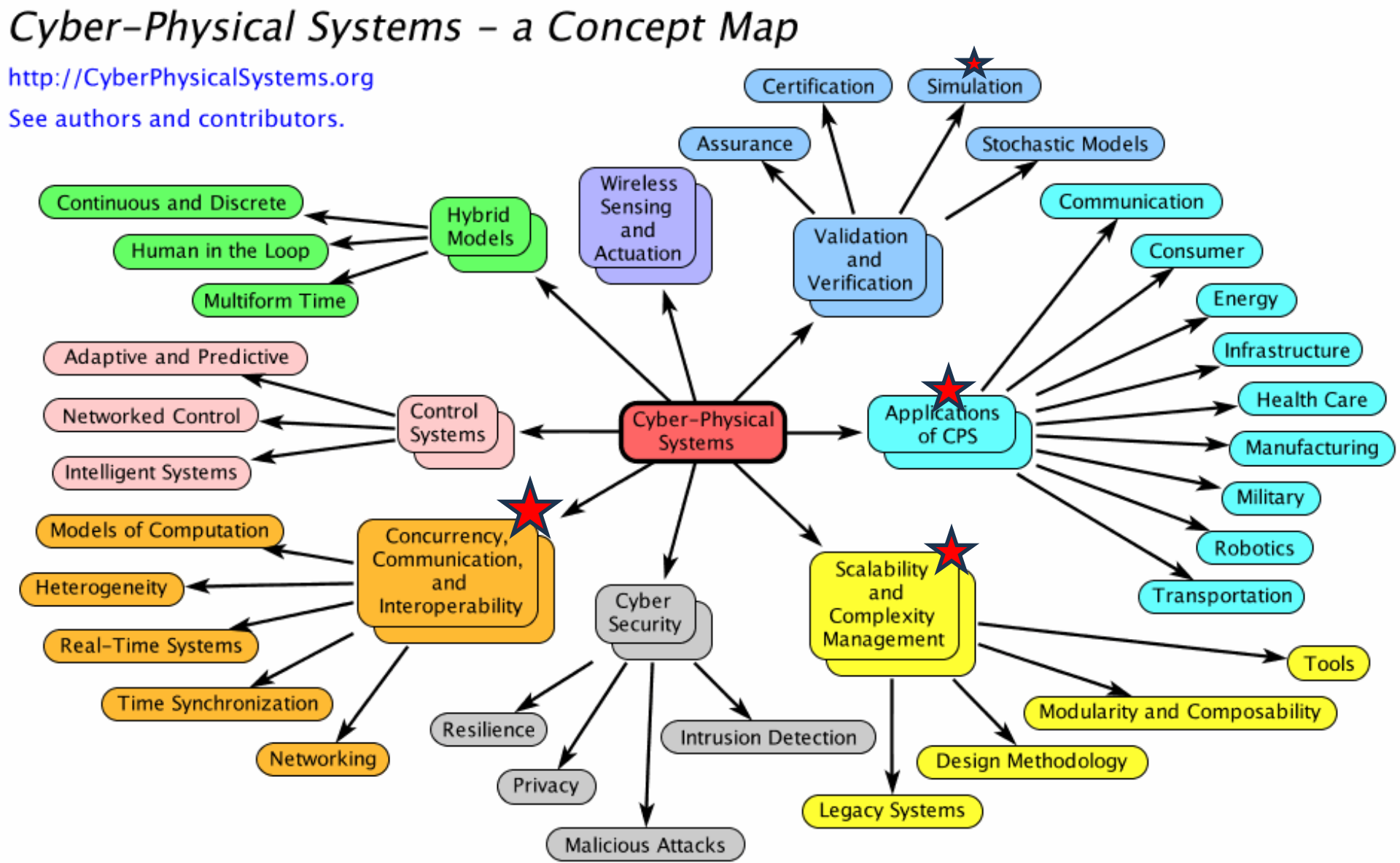


- Domaines connexes : Automatique, Model Predictive Control ("digital twins"), AI/ML

Système de **contrôle embarqué**

- Exemple classique : les systèmes **critiques**
  - Avions, automobiles, centrales nucléaires, appareillage médical, etc.
  - Civil, militaire…
- Mais aussi, plein de produits de consommation:
  - Multimedia (smartphones, lecteur-enregistreur DVD, appareil photo, etc.).
  - Matériel réseau comme les routers, switches, etc.
  - Laves-linges, robots domestiques (e.g. roomba), etc.
- Convergence forte entre les deux
  - Exemples: Software-Defined Vehicle, Drones
    - https://www.renaultgroup.com/wp-content/uploads/2023/04/ppt_event_software_defined_vehicles_vdef.pdf

Système de **contrôle embarqué**



Système de **contrôle embarqué**

- Caractéristiques communes:
  - Systèmes réactifs.
    - Execution *a priori* infinie
  - Exigences non-fonctionnelles, y compris temps-réel
  - Spécification/implantation/V&V compliquées au sens théorique (complexité algorithmique) et au sens de l'ingénierie
    - Spécification: Plusieurs langages/formalismes generalistes (C, Ada,UML) ou dédiés (DSL=Domain Specific Language, comme Simulink, SCADE, AUTOSAR, AADL, SysML, etc.). Utilisation intensive de techniques d'analyse de programmes (vérif. formelle, simulation, etc.)
    - Implantation: Matériel spécifique (contrôleurs contraints en mémoire et vitesse, bus spécifiques, etc.), contraintes de consommation, etc.
  - Les erreurs sont coûteuses (soit en vies humaines, soit en argent).
    - Déterminisme fonctionnel et temporel fortement souhaité.
- Conséquences: Besoins communs dans le processus de développement

Système réactif

- Système qui réagit aux stimuli venant de l'environnement
  - Exécution *a priori* infinie
  - Réactif vs. transformationnel (Harel&Pnueli, 1985)
    - Transformationnel = fonctions de calcul traditionnelles, e.g. l'entraînement d'un réseau de neurones, un filtre appliqué à une photo…
    - Question de point de vue: la plupart des systèmes ont des aspects transformationnels et réactifs
  - Stimuli = événements détectables
    - Appui sur un bouton, arrivée d'un message réseau ou d'une trame vidéo, interruption, dépassement d'un seuil par une variable, etc.
  - Réactions
    - Changements d'état interne, production de signaux, mise à jour des actuateurs, etc.

- Séparation encore mal comprise dans certaines industries
  - E.g. Machine Learning, où l'implantation embarquée, temps réel implique beaucoup de parties manuelles (= opportunité !)

Système réactif

- Les notions de stimulus et réponse doivent être vues en un sens très large:
  - On peut être forcé de faire des calculs intensifs pour déterminer quand un stimulus arrive
    - Logiciel GPS: échantillonner le capteur GPS et les accéléromètres avant de calculer des filtres de Kalman pour savoir quand mettre à jour la position à l'écran et quand donner les annonces.
    - Keyword spotting (Alexa, Siri): traitement de signal et Machine Learning pour détecter un mot-clef (événement)
  - Le temps peut être lui-même un stimulus, e.g. par l'utilisation d'alarmes (timers)
  - Spécification en temps discret ou en temps continu.
- Difficile de trouver aujourd'hui des systèmes informatiques qui ne soient pas réactifs, au moins en partie.
  - E.g. la commande "grep", mais seulement quand elle est prise séparément

- Les réactions peuvent se superposer dans le temps (concurrence) s'interrompre les unes les autres, et impliquer des temporisations (temps-réel)
  - Analyse, vérification, test difficiles
    - Non-déterminisme
    - Beaucoup de configurations possibles

Système temps-réel

- Systèmes réactifs où les réactions doivent être réalisées à des dates et/ou en un temps donné
  - On peut être temps réel sans utiliser des timers !
  - Temps réel dur: toutes les échéances doivent être respectées (e.g. airbag).
  - Temps réel mou: ce n'est pas si grave s'il y a parfois des retards (e.g. box Internet).

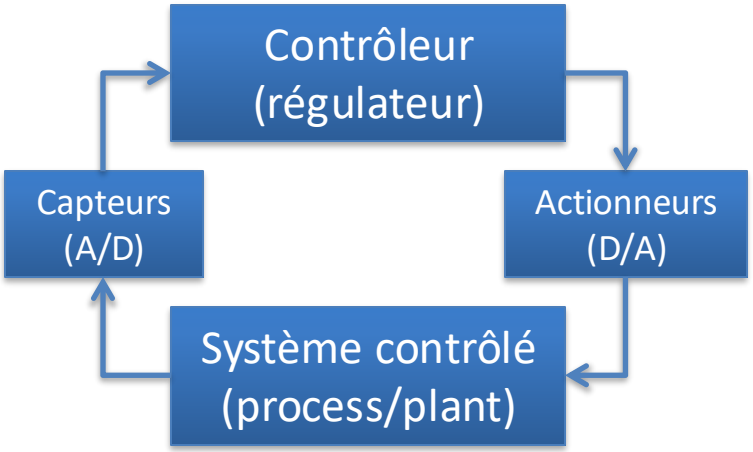
  … mais la limite entre les 2 n'est pas évidente à définir, c'est souvent l'état d'esprit qui compte
    - Des notions de stabilité et de robustesse sont largement employées en ingénierie de systèmes critiques.
    - Notions intermédiaires: « firm real-time »

Système temps-réel

- Jargon temps-réel
  - Tâches = Fonctions calculant les réactions (ou juste des bouts de ces réactions)
  - Arrivée d'une tâche = date où l'exécution d'une tâche **peut** commencer
    - Stimulus arrivé, données disponibles
  - Echéance d'une tâche (deadline) : date après l'arrivée où la tâche doit être finie
    - Échéance manquée (deadline miss) à traiter
  - Types de tâches:
    - Périodiques – arrivent à des intervalles fixes
    - Sporadiques – arrivent avec une distance minimale entre elles
    - Apériodiques – peuvent arriver sans restrictions

Système temps-réel

- Exemple
  - Tâche 1: Drive-by-Wire
    - Tache la plus critique, ne peut pas être arrêtée
    - Temps réel dur
  - Tâche 2: Powertrain control
    - Même criticité que Tache 1
    - Temps réel dur



  - Tâche 3: Autonomie
    - Compréhension de scène, Planification de trajectoire
    - Moins critique que tâches 1, 2 (peut être arrêtée)
    - Temps réel dur, délai d'arrêt
  - Tâche 4: Communication avec le cloud
    - Moins critique que Tache 3
    - Temps réel mou

Ordonnancement temps-réel

- Allocation des ressources aux tâches dans le but d'assurer le respect des échéances
  - Déclenchement/interruption/reprise de tâches
  - Allocation mémoire, dispositifs d'entrée/sortie...
- On peut le faire :
  - En ligne (dynamique) - lors de l'arrivée des stimuli, suivant des politiques (algorithmes) d'ordonnancement généralistes
  - Hors ligne (statique) - dates de départ ou ordre choisies avant exécution (ordonnancement spécifique au système)

…mais la limite entre les 2 n'est pas exactement définie:

- Une politique « en ligne » a des paramètres que l'on peut choisir statiquement avant l'exécution
- Une politique « hors ligne » peut ne pas couvrir des aspects comme l'allocation des bancs mémoire ou même le choix du processeur, qui sont réalisées alors en ligne.

Encore une fois, c'est l'état d'esprit qui compte.

Ordonnancement en ligne

- Implantation événementielle
  - Plusieurs signaux/interruptions peuvent déclencher des calculs. Problèmes de synchronisation
- Algorithmes classiques
  - RM (rate monotonic), FP (fixed priority), EDF (earliest deadline first), DM (deadline monotonic), etc.
- Avantages:
  - Réactions très rapides à des événements prioritaires.
  - Robustesse aux variations temporelles (temps d'exécution, dates d'arrivée des tâches).
- Problèmes:
  - Nécessitent souvent des marges importantes avec les critères d'ordonnançabilité classiques (30% pour RM)
  - Non-déterminisme temporel, plus difficile à vérifier/simuler/tester
  - Exécution conditionnelle difficile à exploiter

Ordonnancement hors ligne

- Ordre et potentiellement date de début des opérations choisies avant l'exécution
- Les événements sont échantillonnés (*e.g.* 1=arrivé, 0=pas arrivé)
- Ordonnancement à base de tables de réservation/scheduling
  - Analyse de temps d'exécution WCET/WCCT
  - Implantation événementielle ou « time-triggered »
    - Time-triggered = tous les calculs sont déclenchés par des timers (**AUTOSAR**, ARINC 653, TTA, FlexRay)

https://www.researchgate.net/publication/273163359_Dynamic_Task_Scheduling_on_Multicore_Automotive_ECUs

Ordonnancement hors ligne

- Avantages:
  - Déterminisme temporel, plus facile a vérifier/tester/certifier
  - Pas besoin de marges temporelles (utilisation 100%)
  - Prise en compte facile des conditions d'exécution
- Désavantages:
  - Allocation statique des ressources, au pire cas
    - OK pour systèmes très critiques
  - Vitesse de réaction dépendant des réservations.
  - Implantation plus compliquée, car il y a plus de choses à synthétiser (e.g. la table d'ordonnancement)

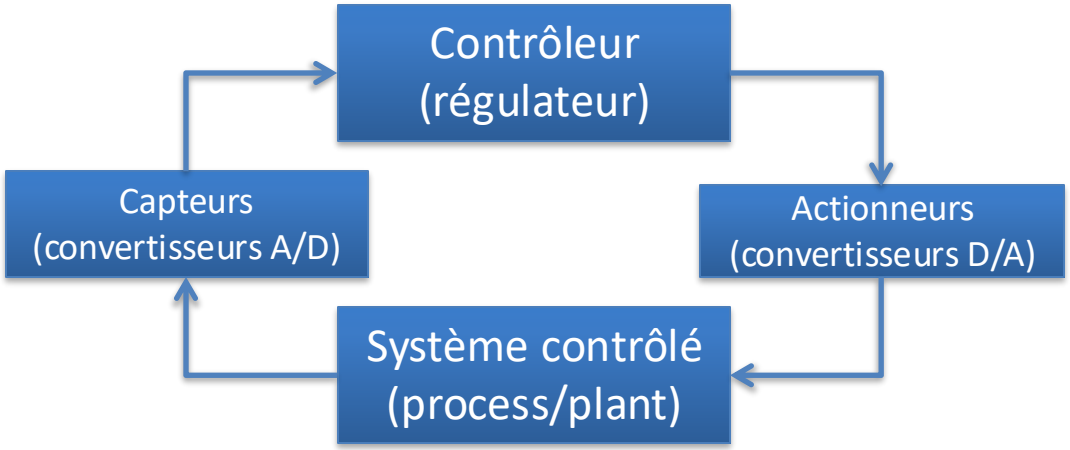
- Ordonnancement optimal = NP complet
  - Heuristiques = algorithmes approchés (non-optimaux) éprouvés en pratique (de type « compilation »)
    - Comme les politiques "en ligne"
  - Hors ligne/en ligne

Programmation synchrone

- Bases formelles
- Langage **flot de données** Lustre/Heptagon
- Compilation et génération de code

Programmation synchrone

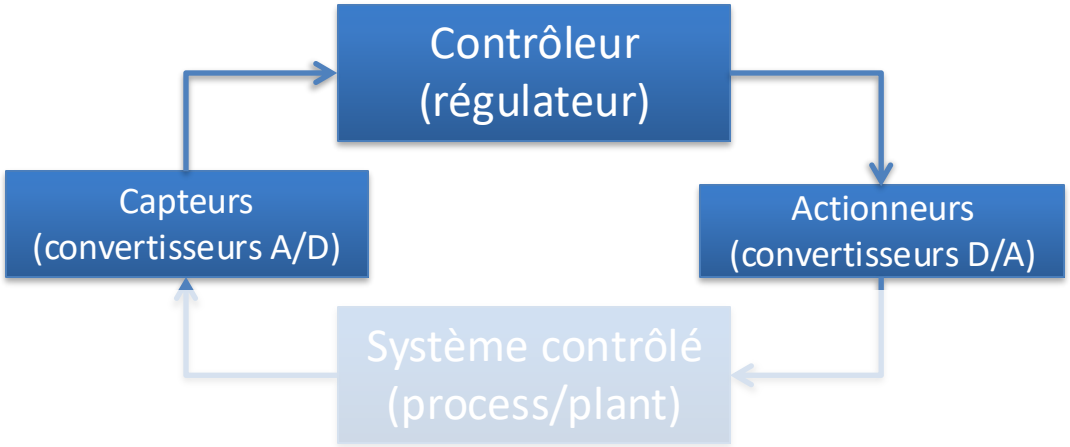
- Langages pour systèmes de contrôle-commande



  - Autres utilisations possible (signal processing, ML,…)
    - SP, ML souvent utilisés dans des systèmes embarqués

Programmation synchrone

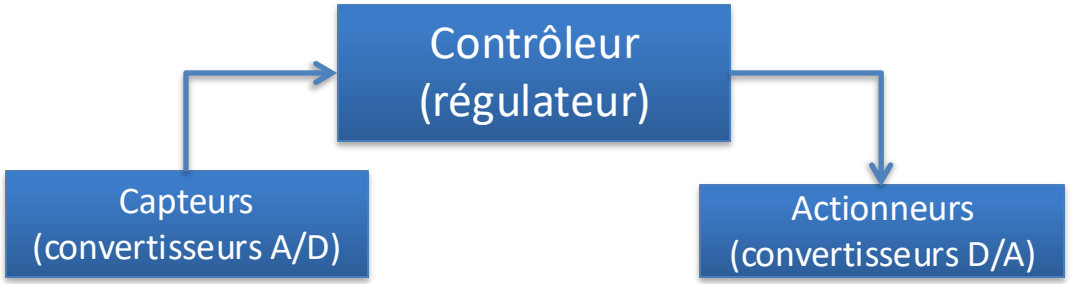
- Langages pour systèmes de contrôle-commande



  - Autres utilisations possible (signal processing, ML,…)
    - SP, ML souvent utilisés dans des systèmes embarqués

Programmation synchrone

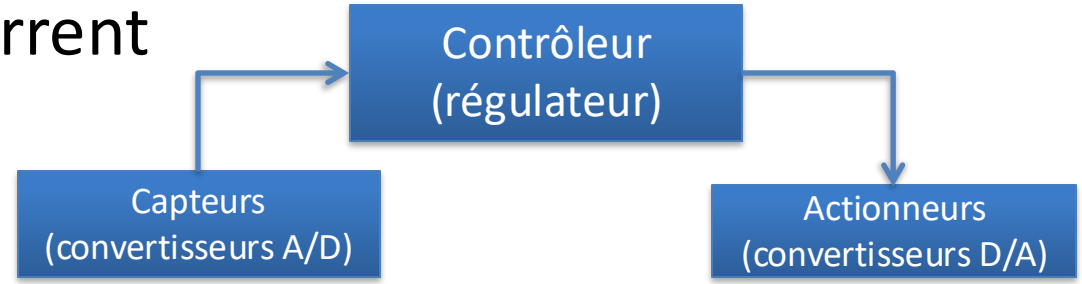
- Langages pour systèmes de contrôle-commande
  - Globalement séquentiel – notion de cycle d'exécution

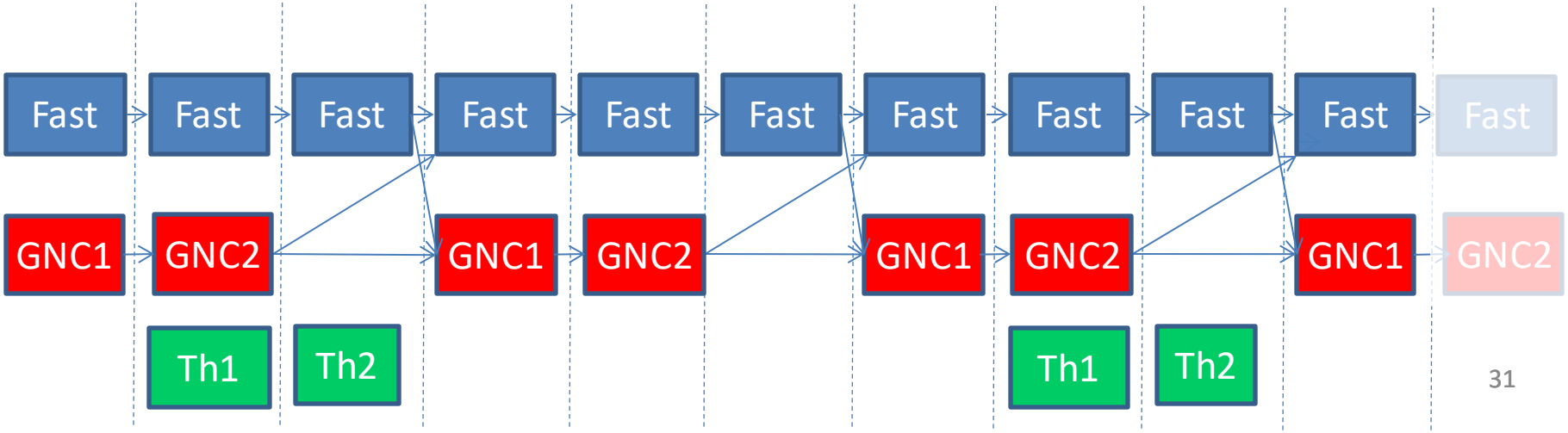



```
for(;;) {
  read_inputs() ;
  compute() ;
  write_outputs() ;
}
```

Programmation synchrone

- Langages pour systèmes de contrôle-commande
  - Globalement séquentiel
  - Localement concurrent
  - Déterministe





Pourquoi synchrone ?

- Programmer ces systèmes est coûteux…
  - Coût très important : $200-1000/LoC (déjà en 2014)
    - https://sel4.systems/Info/Docs/GD-NICTA-whitepaper.pdf
    - https://en.wikipedia.org/wiki/Evaluation_Assurance_Level#EAL6:_Semiformally_Verified_Design_and_Tested

- … car on a besoin de garanties de correction :
  - Les erreurs sont coûteuses
    - Argent: Vol 501 d'Ariane 5
    - Vies humaines: Bug Toyota, Missiles Patriot pendant la guerre d'Iraq (1991), accidents de Tesla, etc.
      - Attention – tout bug n'est pas dû au logiciel - dans le cas du MCAS sur les Boeing 737MAX, le logiciel a bien fait ce qui était prévu, les problèmes venant de l'ingénierie système
  - Processus de développement bien établis
  - Certification


Accident mortel à Paris : défaillance, bug, boîte noire… Tesla au cœur de l'enquête


Tesla rappelle 2 millions de véhicules pour un problème sur Autopilot

Programmation synchrone

- Programmer ces systèmes est difficile :
  - Logiciels de très grande taille
    - Centaines de milliers de lignes de code, en rapide augmentation, notamment par les aspects IA/ML, middleware, user experience…
  - Logiciels complexes:
    - Réactif, temps réel
    - Concurrent
    - Architectures dédiées, multi-/many-core
- Approches spécifiques
  - Forte structuration du code pour:
    - **Faciliter design**/debug/test
      - **Déterminisme fonctionnel** souhaité
    - Garantir la traçabilité, la reproductibilité
    - Obtenir des **propriétés par construction**
  - Méthodes d'implantation et outils qualifiables

Programmation synchrone

- Langages synchrones
  - Programmation des aspects fonctionnels d'une application
    - Fonctionnel = ce qui est calculé
  - Programmation par composants
  - Concurrence exposée **facilitant le design**
    - Implantation = allocation, ordonnancement, génération de code
    - Identification du parallélisme potentiel
  - **Déterminisme fonctionnel**
    - Facile à vérifier, déboguer, tester
    - Même résultat entre analyse, simulations, implantation

Programmation synchrone

- Forme de programmation disciplinée
  - Pas de code à exécution infinie
  - Pas de variable non-initialisée
  - Pas de double initialisation
  - Pas de « race condition »

  …
- Sémantique formelle
  - Vérification et test formels
    - Modèles formels : circuits synchrones et automates à états finis
    - **Calculs d'horloges = vérifications de correction à faible coût computationnel, intégrées aux compilateurs**

Programmation synchrone

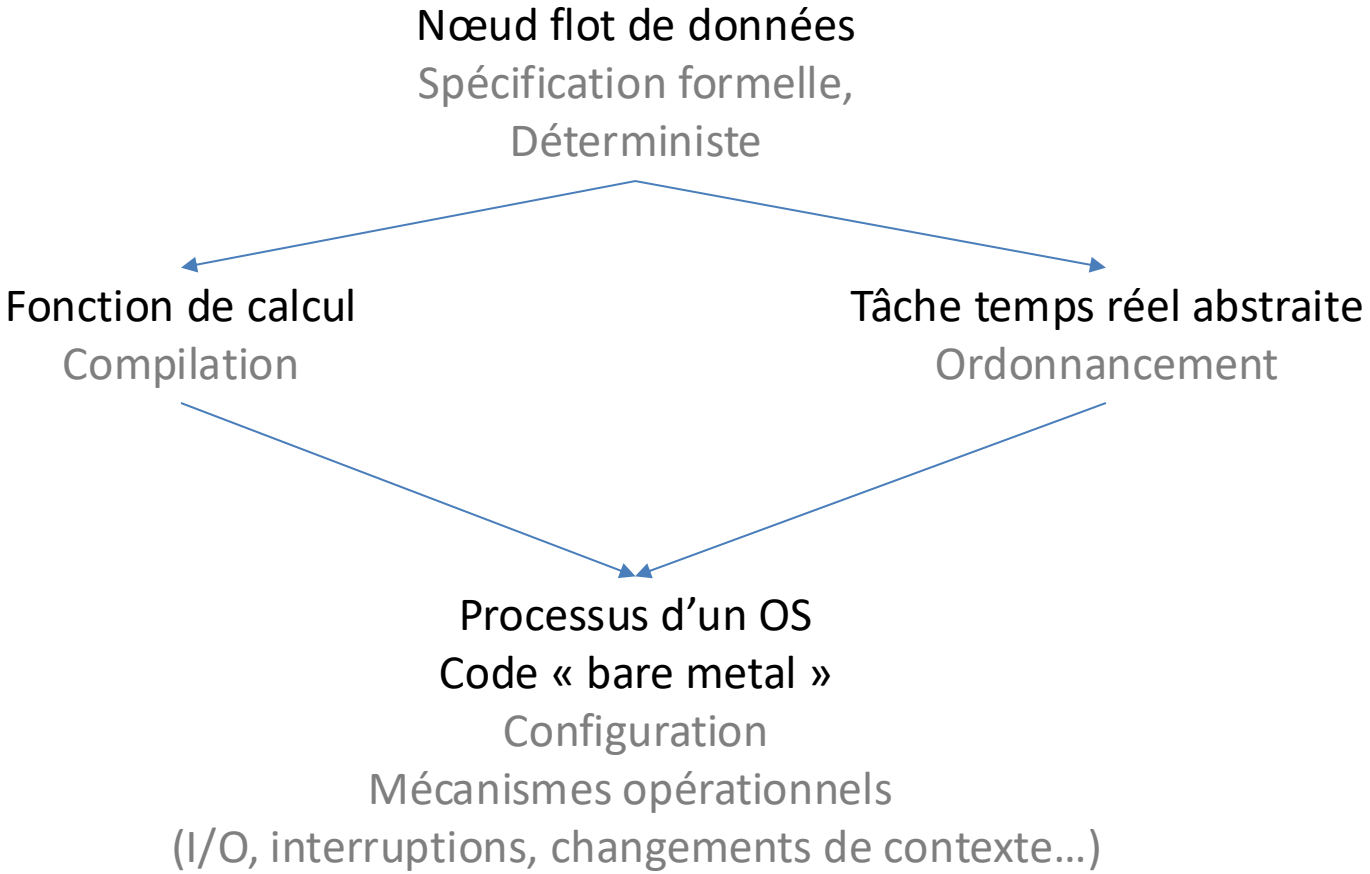
- Langages et formalismes synchrones
  - Flot de données
    - Scade, sous-ensembles "sûrs" de Simulink
    - Lustre, **Heptagon**, Signal
  - Flot de contrôle
    - Esterel, Quartz
    - PsyC

Programmation synchrone flot de données

- Flot de données = style de présentation des algorithmes
  - Naturel en:
    - Systèmes embarqués critiques (Tâches, Scade/Lustre)
    - Automatique (Simulink)
    - Algorithmique multimédia/IA/Big Data
    - Conception de circuits digitaux synchrones (netlists)
  - Algorithme = graphe dirigé
    - Nœuds = fonctions de calcul et mémoires.
      - Les nœuds ont des ports d'entrée et de sortie nommés
    - Arcs = passages de valeurs d'un nœud à l'autre, permettant les calculs.

Programmation synchrone flot de données

- Intuition: **tâche temps réel = nœud flot de données**



Le langage Lustre/Heptagon

- La brique de base: la fonction de calcul

$(o_1,\ldots,o_m) = f(i_1,\ldots,i_n)$

$o = f(i_1,\ldots,i_n)$

  - Cycliquement, attend une valeur sur chacune des variables d'entrée et calcule une valeur sur chacune des variables de sortie. Aucune valeur ne doit être perdue ou créée.
  - Hypothèses :
    - Aucun effet de bord sur le calcul d'autres fonctions.
    - Pas de mémoire entre appels successifs.
    - Temps d'exécution borné

Le langage Lustre/Heptagon

- Exemple :

$$o = i_1 + i_2$$

  - Chronogramme

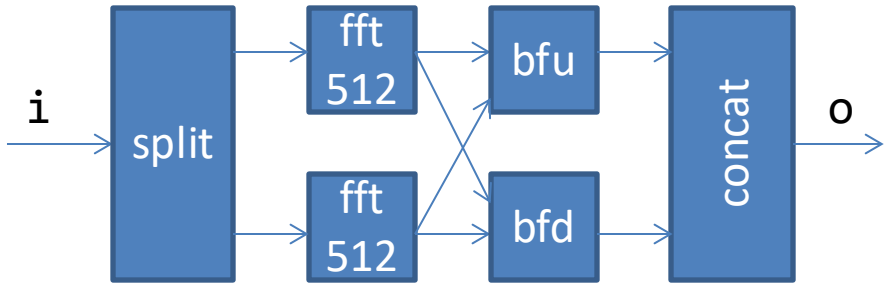
cycle	0	1	2	3	4	5	6	…
i_1	2	7	12	17		22	27	…
i_2	14	12	10	8		6	4	…
o	16	19	22	25		28	31	…

  - Une variable peut être présente ou absente durant un cycle
    - Synchronisation nécessaire entre variables

Le langage Lustre/Heptagon

- On peut composer les fonctions Heptagon pour calculer des fonctions (au sens mathématique) compliquées:

```
fft1024:
        (i1,i2) = split(i) ;
             v1 = fft512(i1) ;
             v2 = fft512(i2) ;
             o1 = bfu(v1,v2) ;
             o2 = bfd(v1,v2) ;
              o = concat(o1,o2) ;
```



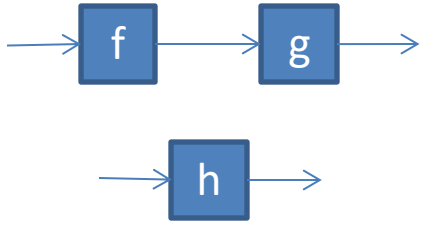
  - Une fonction peut être appelée (instanciée) plusieurs fois
  - Chaque port d'entrée est connecté à exactement un port de sortie (pour éviter les ambigüités)
  - Mais un port de sortie peut être connecté à 0, 1, ou plusieurs ports d'entrée

Le langage Lustre/Heptagon

- Pas de séquencement implicite
  - Dépendances de données explicites ➡ Concurrence

```
y=f(x);
z=g(y);
u=h(v);
```

Ordre d'exécution fixé en C



Plusieurs ordres séquentiels possibles en Heptagon :

f;g;h;
f;h;g;
h;f;g;

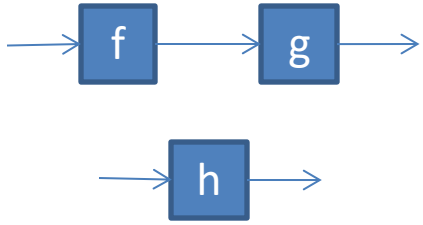
  - Ordre partiel ➡ Tout ordre total le contenant est une exécution mono-processeur correcte.
    - Important : absence d'effets de bords des noeuds !

Le langage Lustre/Heptagon

- Pas de séquencement implicite
  - Dépendances de données explicites ➡ Concurrence

```
y=f(x);
z=g(y);
u=h(v);
```

Ordre d'exécution fixé en C



Plusieurs ordres séquentiels possibles en Heptagon :

f;g;h;
f;h;g;
h;f;g;

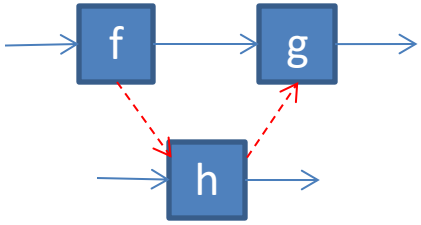
  - Ordre partiel ➡ Tout ordre total le contenant est une exécution mono-processeur correcte.
    - Degré de liberté pour la génération de code efficace.
      - Les compilateurs (GCC, LLVM…), font de même: analyse de données, SSA.

Le langage Lustre/Heptagon

- Pas de séquencement implicite
  - Dépendances de données explicites ➡ Concurrence

```
y=f(x);
z=g(y);
u=h(v);
```

Ordre d'exécution fixé en C



Plusieurs ordres possibles, mais on peut ajouter des dépendances…

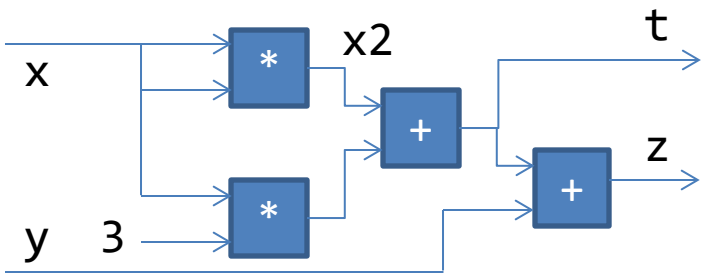
~~f;g;h;~~
f;h;g;
~~h;f;g;~~

  - Ordre partiel ➡ Tout ordre total le contenant est une exécution mono-processeur correcte.
    - Degré de liberté pour la génération de code efficace.
      - Les compilateurs (GCC, LLVM…), font de même: analyse de données, SSA.

Le langage Lustre/Heptagon

- Expressions flot de données

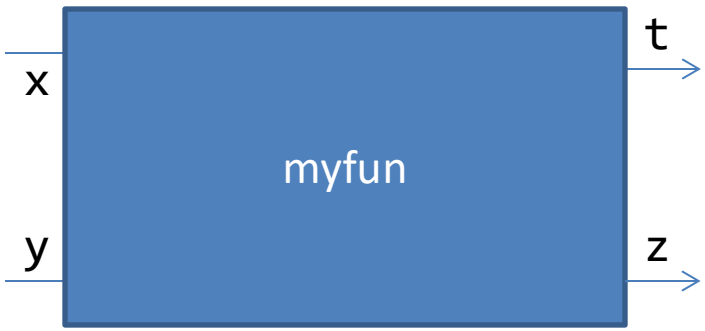
```
x2 = x*x ;
t = x2 + 3*x ;
z = t + y ;
```



Le langage Lustre/Heptagon

- Hiérarchie

```
fun myfun (x:int;y:int) returns (z:int;t:int)
var x2 : int;
let
  x2 = x*x ;
  t = x2 + 3*x ;
  z = t + y ;
tel
```

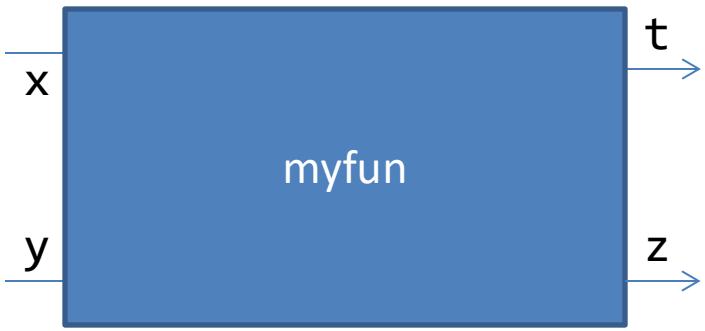


Le langage Lustre/Heptagon

- Hiérarchie

```
fun myfun (x:int;y:int) returns (z:int;t:int)
var x2 : int;
let
  x2 = x*x ;
  t = x2 + 3*x ;
  z = t + y ;
tel
```

```
...
(t,z) = myfun(x,y) ;
...
```

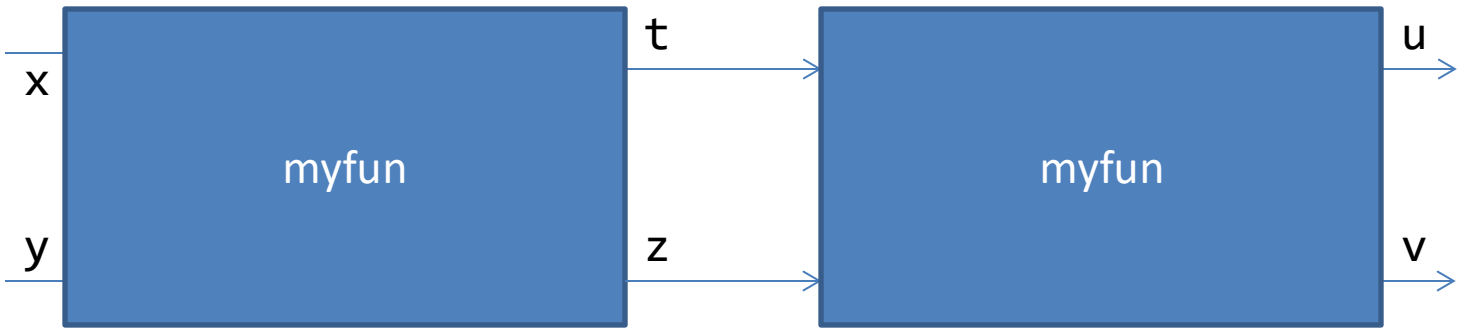


Le langage Lustre/Heptagon

- Hiérarchie

```
fun myfun (x:int;y:int) returns (z:int;t:int)
var x2 : int;
let
  x2 = x*x ;
  t = x2 + 3*x ;
  z = t + y ;
tel
```

```
...
(t,z) = myfun(x,y) ;
(u,v) = myfun(t,z) ;
...
```



Le langage Lustre/Heptagon

- Typage strict des données
  - Pas de polymorphisme, même pour l'arithmétique
    - 1 + 3 : int
    - 1. +. 3. : float
  - Types de données élementaires :
    - bool : true, false
    - int : opérateurs +,-,*,/
    - float : opérateurs +., -., *., /.

Le langage Lustre/Heptagon

- Ce que l'on a défini déjà :
  - Appel de fonctions
  - Hiérarchie
  - Pas d'état
  - Pas de contrôle
- Déjà certains systèmes industriels peuvent être programmés
  - Contrôle proportionnel (e.g. gyropode/Segway)
  - FFT (en fonction de la présentation des données)

Le langage Lustre/Heptagon

- État d'un programme – instruction fby

```
z = x fby y ;
```

$z_0 = x_0$

$z_{n+1} = y_n$, $n \geq 0$

– Intuition – code globalement séquentiel

```
z = x ;    /* initialisation */
for(cycle=0;;cycle++){
  …        /* calcul du cycle */
  z = y ;  /* passage au cycle suivant */
}
```

Le langage Lustre/Heptagon

- État d'un programme – instruction fby

```
z = x fby y ;
```

$z_0 = x_0$

$z_{n+1} = y_n$, $n \geq 0$

– Chronogramme

cycle	0	1	2	3	4	5	6	…
x	1	2	3	4	5	6	7	…
y	10	9	8	7	6	5	4	…
z	1	10	9	8	7	6	5	…

Le langage Lustre/Heptagon

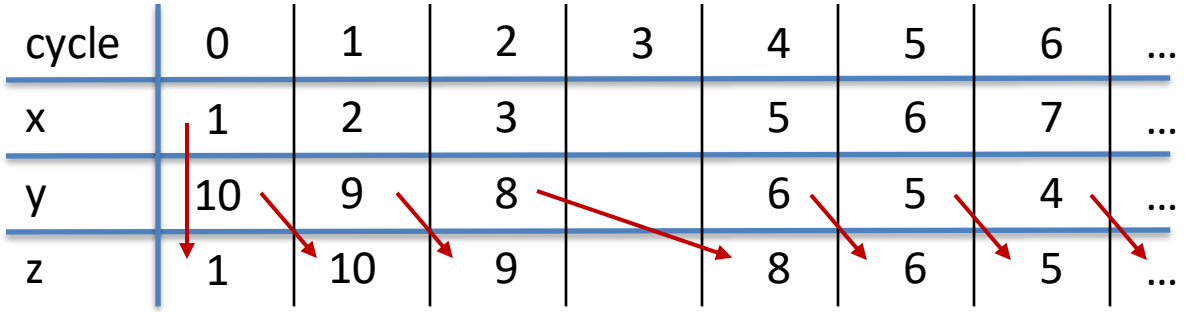
- État d'un programme – instruction fby

```
z = x fby y ;
```

$z_0 = x_0$

$z_{n+1} = y_n$, $n \geq 0$

– Chronogramme



Le langage Lustre/Heptagon

- Un compteur

```
node counter () returns (cnt:int)
var pre_cnt : int ;
let
  pre_cnt = cnt + 1 ;
  cnt = 0 fby pre_cnt ;
tel
```

cycle	0	1	2	3	4	5	6	…
cnt	0	1	2	3	4	5	6	…

Le langage Lustre/Heptagon

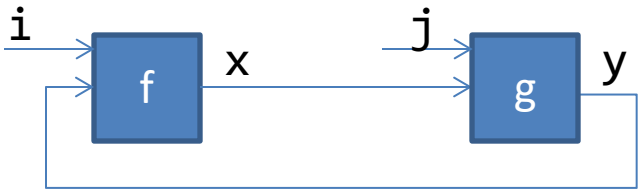
- Un compteur – utilisation d'expressions

```
node counter () returns (cnt:int)
let
  cnt = 0 fby (cnt+1) ;
tel
```

cycle	0	1	2	3	4	5	6	…
cnt	0	1	2	3	4	5	6	…

Le langage Lustre/Heptagon

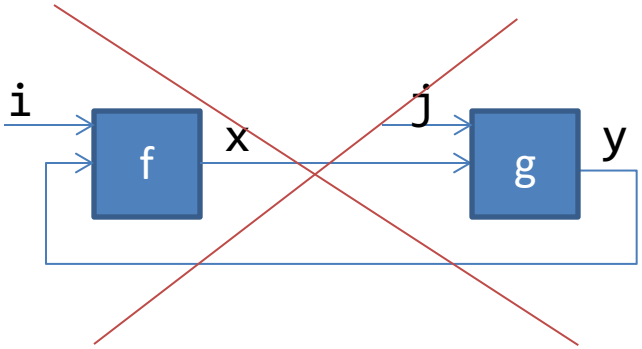
- Causalité (dépendances dans l'exécution)
  - Un programme incorrect - le calcul n'avance pas
    - Intuition : qui exécuter en premier ?

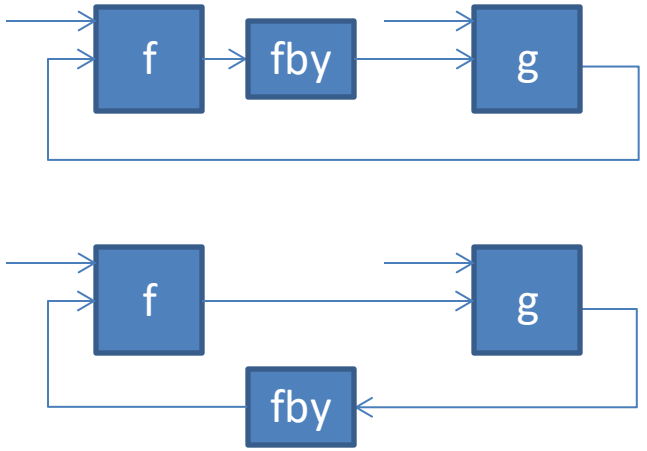



```
node cycle(i:int;j:int) returns (y:int)
var x : int ;
let
  x = f(i,y) ;
  y = g(j,x) ;
tel
```

Le langage Lustre/Heptagon

- Causalité (dépendances dans l'exécution)
  - Versions correctes minimales





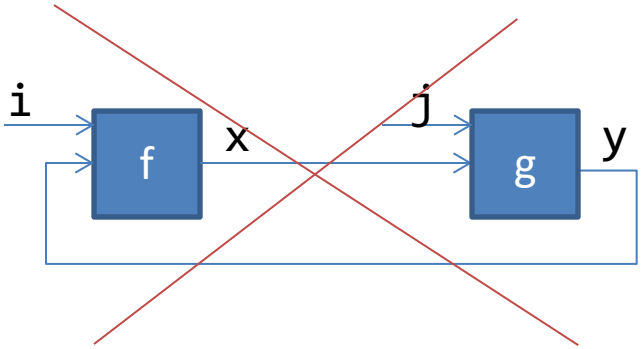

```
node cycle(i:int;j:int) returns (y:int)
var x : int ;
let
  x = f(i,y) ;
  y = g(j,x) ;
tel
```

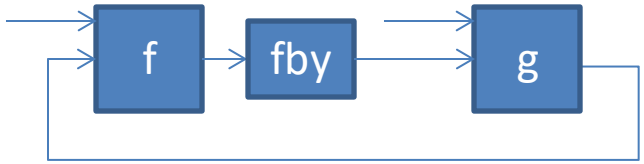
```
let
  x = f(i,y) ;
  y = g(j,0 fby x) ;
tel
```

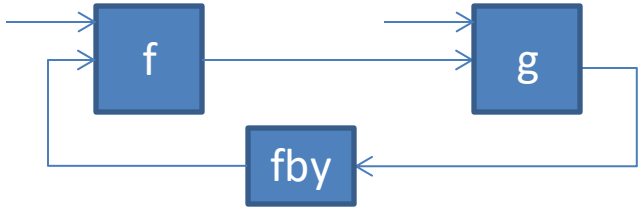
```
let
  x = f(i,0 fby y) ;
  y = g(j,x) ;
tel
```

Le langage Lustre/Heptagon

- Causalité (dépendances dans l'exécution)
  - Versions correctes minimales



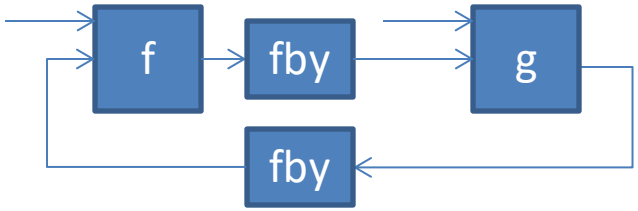


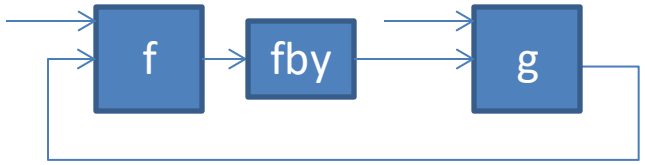


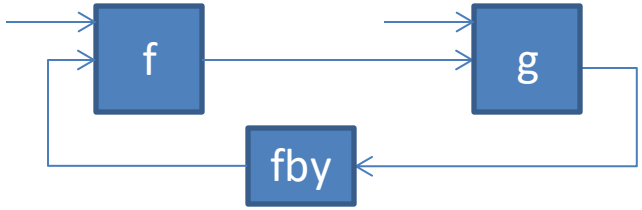
  - Règle assurant le déterminisme
    - **Tout cycle du graphe doit contenir un fby**

Le langage Lustre/Heptagon

- Causalité (dépendances dans l'exécution)
  - Versions correctes



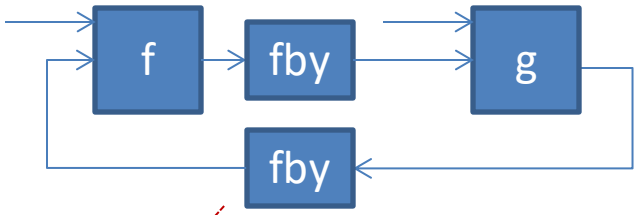


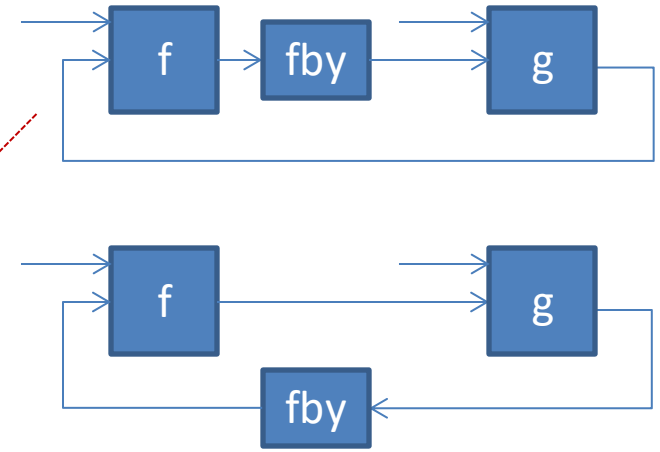


  - Règle assurant le déterminisme
    - **Tout cycle du graphe doit contenir un fby**

Le langage Lustre/Heptagon

- Causalité (dépendances dans l'exécution)
  - Intuition impérative






```
x1 = 0 ; y1 = 0 ;
for(;;){
  y = g(j,x1) ;
  x = f(i,y1) ;
  x1 = x ; y1 = y ;
}
```

```
x = 0 ;
for(;;){
  y = g(j,x) ;
  x = f(i,y) ;
}
```

```
y = 0 ;
for(;;){
  x = f(i,y) ;
  y = g(j,x) ;
}
```

Le langage Lustre/Heptagon

- Ce que l'on a défini déjà :
  - Appel de fonctions
  - Hiérarchie
  - État
  - Pas de contrôle
- Beaucoup de systèmes industriels peuvent être programmés
  - Sans les aspects « système » comme la gestion d'erreurs
  - E.g. PID (proportional/integral/derivative) controller

Le compilateur Heptagon

- Compilation d'un programme :

first.ept

```
fun myfun (x:int;y:int) returns (z:int;t:int)
var x2 : int;
let
  x2 = x*x ;
  t = x2 + 3*x ;
  z= t + y ;
tel

node sum2(i:int) returns (o:int)
let
  o = i + (0 fby i) ;
Tel

node counter () returns (cnt:int)
let
  cnt = 0 fby (cnt+1) ;
tel
```

Le compilateur Heptagon

- Compilation d'un programme :

  heptc first.ept

  – Vérifie la correction du programme

  – Construit des représentations intermédiaires

  heptc –target c first.ept

  – All the above

  – Crée le répertoire first_c et y génère le code C associé aux fonctions et nœuds

Le compilateur Heptagon

- Compilation de first.ept

first_types.h
first_types.c
first.h
first.c

```
fun myfun (x:int;y:int) returns (z:int;t:int)
var x2 : int;
let
  x2 = x*x ;
  t = x2 + 3*x ;
  z= t + y ;
tel
```

```
void First__myfun_step(
       int x, int y,
       First__myfun_out* _out) ;
```

```
typedef struct First__myfun_out {
  int z;
  int t;
} First__myfun_out;
```

Le compilateur Heptagon

- Structure d'un identifiant généré :

first_types.h
first_types.c
first.h
first.c

```
fun myfun (x:int;y:int) returns (z:int;t:int)
var x2 : int;
let
  x2 = x*x ;
  t = x2 + 3*x ;
  z= t + y ;
tel
```

First__myfun_step

Nom du fichier source, capitalisé

Nom de la fonction ou du noeud

Fonction de l'objet

First__myfun_out

Le compilateur Heptagon

- Compilation de first.ept

first_types.h
first_types.c
first.h
first.c

```
node sum2(i:int) returns (o:int)
let
  o = i + (0 fby i) ;
tel
```

```
void First__sum2_step(
    int i,
    First__sum2_out* _out,
    First__sum2_mem* self);
void First__sum2_reset(
    First__sum2_mem* self);

typedef struct First__sum2_mem {
  int v; // the state of the fby
} First__sum2_mem;
typedef struct First__sum2_out {
  int o;
} First__sum2_out;
```

Génération de code exécutable

- Exécution du code pour une fonction :

```
void main() {
  int x,y ;             /* Allocation des entrées */
  First__myfun_out o ; /* Allocation des sorties */
  for(;;) {             /* Boucle infinie */
    printf("Inputs:"); scanf("%d%d",&x,&y) ; /* Lecture des entrées */
    First__myfun_step(x,y,&o) ;              /* Calculs */
    printf("Result: z=%d t=%d\n",o.z,o.t) ;  /* Ecriture des sorties */
  }
}
```

Génération de code exécutable

- Exécution du code pour un nœud :

```
void main() {
  int i ;              /* Allocation des entrées */
  First__sum2_out o ; /* Allocation des sorties */
  First__sum2_mem s ; /* Allocation d l'état */
  First__sum2_reset(&s) ; /* Initialisation de l'état */
  for(;;) {            /* Boucle infinie */
    printf("Inputs:"); scanf("%d",&i) ; /* Lecture des entrées */
    First__sum2_step(i,&o,&s) ;         /* Calculs */
    printf("Result: o=%d\n",o.o) ;      /* Ecriture des sorties */
  }
}
```

- Remarque : Allocation statique de toutes les variables d'entrée et de sortie !
  - La consommation mémoire est contrôlée

Génération de code exécutable

- Déclenchement de cycles de calcul :

```
void main() {
  First__counter_out o ; /* Allocation des sorties */
  First__counter_mem s ; /* Allocation d l'état */
  First__counter_reset(&s) ; /* Initialisation de l'état */
  for(;;) {              /* Boucle infinie */
    wait_trigger() ; /* A vous d'en choisir un (timer, input, etc.) */
    First__counter_step(&o,&s) ;
    printf(" Result: cnt=%d\n",o.cnt) ;
  }
}
```

Préparation du TP

- Objectifs:
  - Installation de Heptagon
    - Suivez les instructions de https://gitlab.inria.fr/synchrone/heptagon
      - Installation avec OPAM, sans simulateur graphique
      - Appelez-moi si vous avez des problèmes
    - Documentation sur le langage : https://gitlab.inria.fr/synchrone/heptagon/-/blob/master/manual/heptagon-manual.pdf
      - Posez-moi des questions !
  - Écriture des premiers programmes
    - Compilation
    - Exécution du code

Préparation du TP

- Exemples à programmer (transparent 62) :
  - Objectif 1 : myfun
  - Objectif 2 : sum2
  - Objectif 3 : counter
- Le fichier source unique étant donné (transparent 62), il faut pour chaque exemple :
  - Le compiler vers du code C en utilisant le compilateur Heptagon
  - Écrire la fonction main (en C)
    - Pour "counter", "wait_trigger" doit être implanté comme une attente de 1 seconde, en utilisant la fonction POSIX usleep
    - Pour les autres pas besoin de usleep, la lecture des entrées depuis le clavier crée l'attente
  - Compiler le code C vers du code exécutable
  - Exécution séquentielle du binaire
  - L'ensemble des étapes de compilation doivent être réalisées soit à l'aide d'un script bash nommé "compile.sh", soit à l'aide d'un Makefile

Préparation du TP

- Pour chaque objectif X, créer un répertoire tp1-objX-NOM (ou NOM est le nom de l'étudiant)
  - Exemple: tp1-obj2-potop
  - Placer dedans le code Heptagon, le code C écrit à la main (main.c) et le Makefile (ou script de compilation)

- Attention : la compilation d'un fichier abc.ept produit un répertoire abc_c contenant le code C généré.
  - Ce répertoire est écrasé à chaque compilation
  - Ne pas y placer du code C écrit manuellement
    - Ne jamais modifier un fichier C généré (e.g. pour y loger la fonction main)

Préparation du TP

- Comment rendre un objectif fini :
  - M'appeler, et me le montrer sur ordinateur
  - Ensuite archiver le répertoire en format .tar.gz ou .zip (avec le même nom que le répertoire) et m'envoyer l'archive par mél (dumitru.potop@inria.fr)