

# Une approche synchrone à la conception de systèmes embarqués temps réel

Dumitru Potop-Butucaru

[dumitru.potop@inria.fr](mailto:dumitru.potop@inria.fr)

cours EIDD, 2023

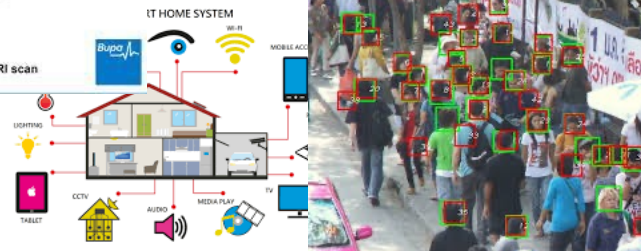
# Contenu de ce cours



## Système embarqué



A person having an MRI scan



# Contenu de ce cours

**Système embarqué**

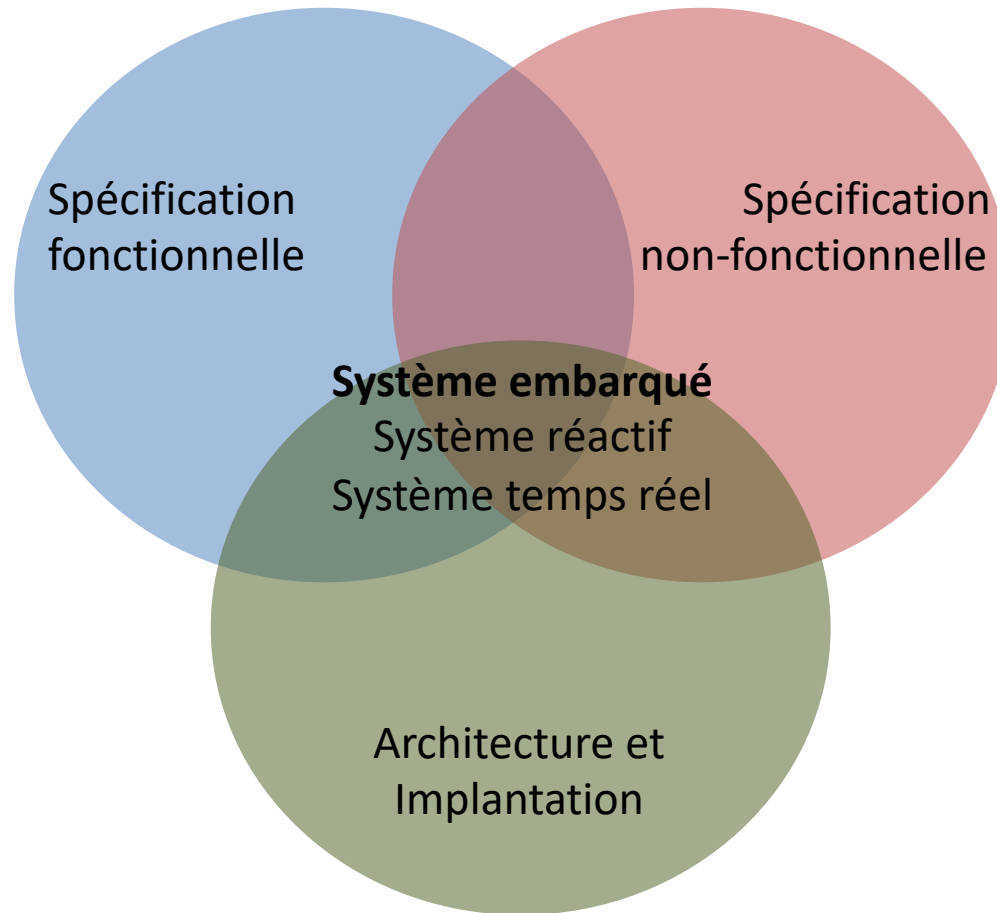
Système réactif

Système temps réel

# Contenu de ce cours

Langages dédiés

- Synchrones
- LET = logical execution time



Temps réel

Mémoire

Partitionnement...

Architectures dédiées

ARINC 653

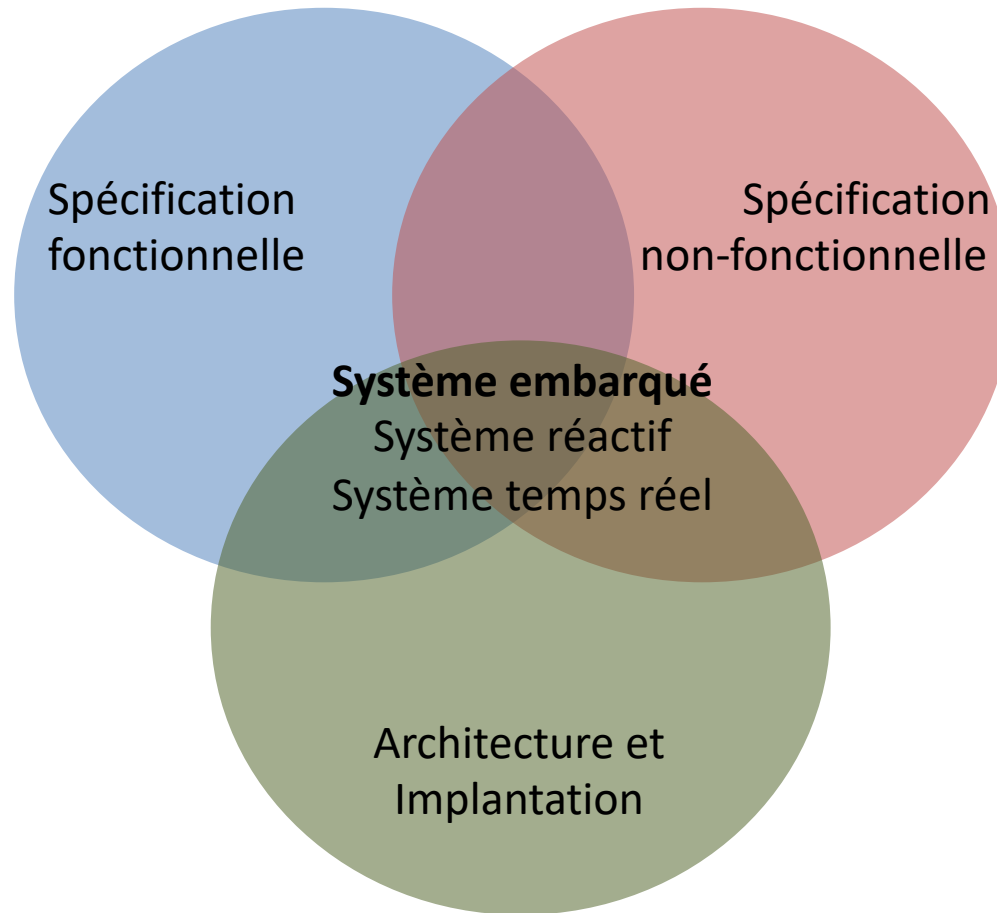
Ordonnancement (en ligne, hors ligne)

Génération de code

# Contenu de ce cours

Langages dédiés

- Synchrones
- LET = logical execution time



Temps réel

Mémoire

Partitionnement...

Approche "hands-on"

- Spécification
- Génération de code
- Exécution sur PC et cartes Raspberry Pi

Architectures dédiées

ARINC 653

Ordonnancement (en ligne, hors ligne)

Génération de code

# Contenu de ce cours – le bas niveau

- C'est un cours intégratif !
  - Connaissances préalables 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 (pas de gdb ou xcode sur cible embarquée, notamment en présence de concurrence)
- Outils open-source
  - Heptagon, gcc (compilateurs), RPi653 (OS)
  - Situation proche d'un environnement de production, où les outils très stables (e.g. gcc) et les outils moins stables se cotoient
    - Vous êtes de futurs ingénieurs !

# Points particuliers

- Flot de données
  - Je vous incite fortement à bien vous approprier ce paradigme de programmation
    - Domaines d'utilisation:
      - Temps réel (Lustre/Scade/Heptagon...)
      - Traitement de signal (Simulink...)
      - 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 ouvert
- Programmation bas-niveau et système
  - Le besoin existera toujours, même si les plates-formes changent
  - Exemple : comment mettre un code ML sur une carte embarquée ?

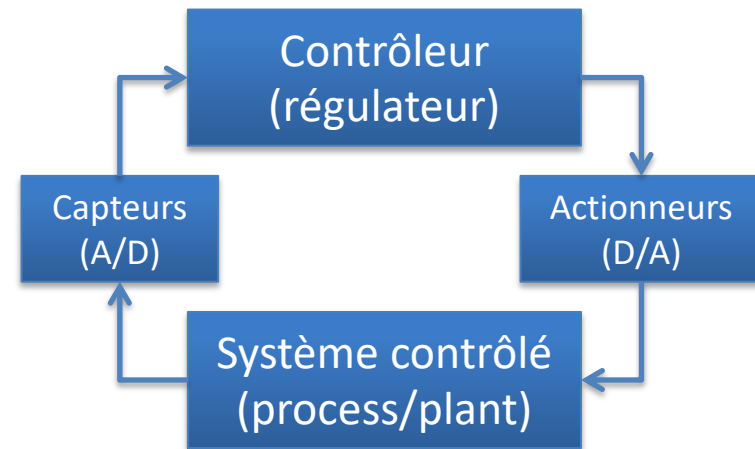
# Evaluation

- 2/3 = Contrôle continu, sur le TP
  - 1 séance = 1 cours + 1 TP
  - Travail individuel, puis (quand on passe sur carte Raspberry Pi) en binôme
  - Évaluation par problème résolu
    - Appelez-moi durant le TP dès qu'un problème est résolu, pour que je le note
      - Une fois le problème rendu, quand il y a du code à envoyer, envoi par mél à mon adresse
    - N'hésitez pas à m'appeler rapidement si vous êtes bloqués
      - C'est ce qui justifie ma présence en salle lors des TP
  - Le rendu est attendu, sauf exception notée, le soir de 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)
- 1/3 = Examen
  - Programmation faisant intervenir les aspects théoriques
- Rattrapage – le tout (continu+examen) condensé sur 1-2 jours
  - Déconseillé



# 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



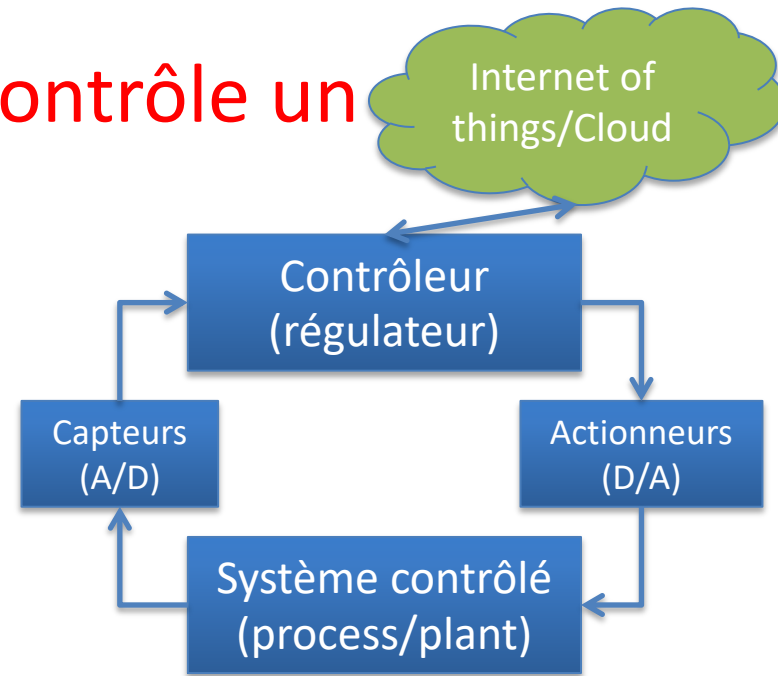
# Edge computing

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

- **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

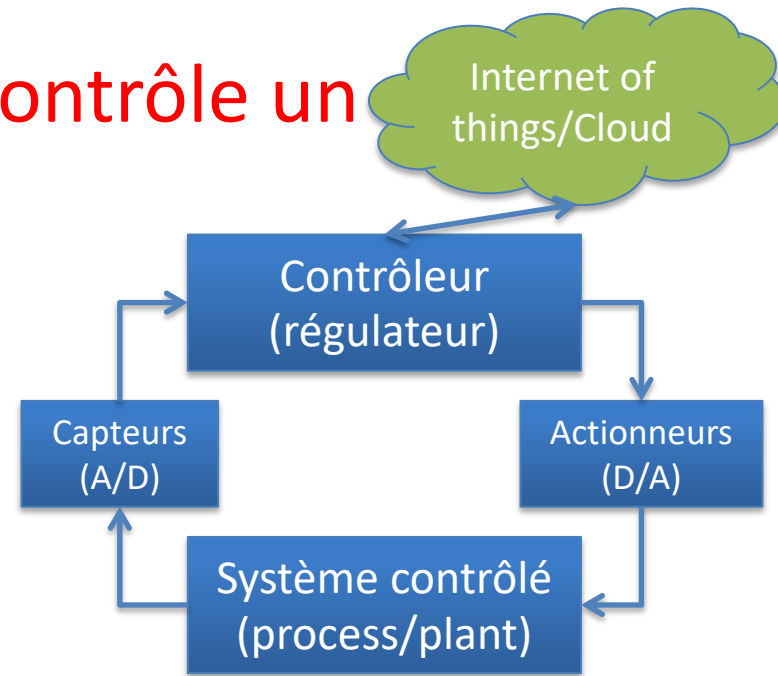
## Edge computing

- **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**

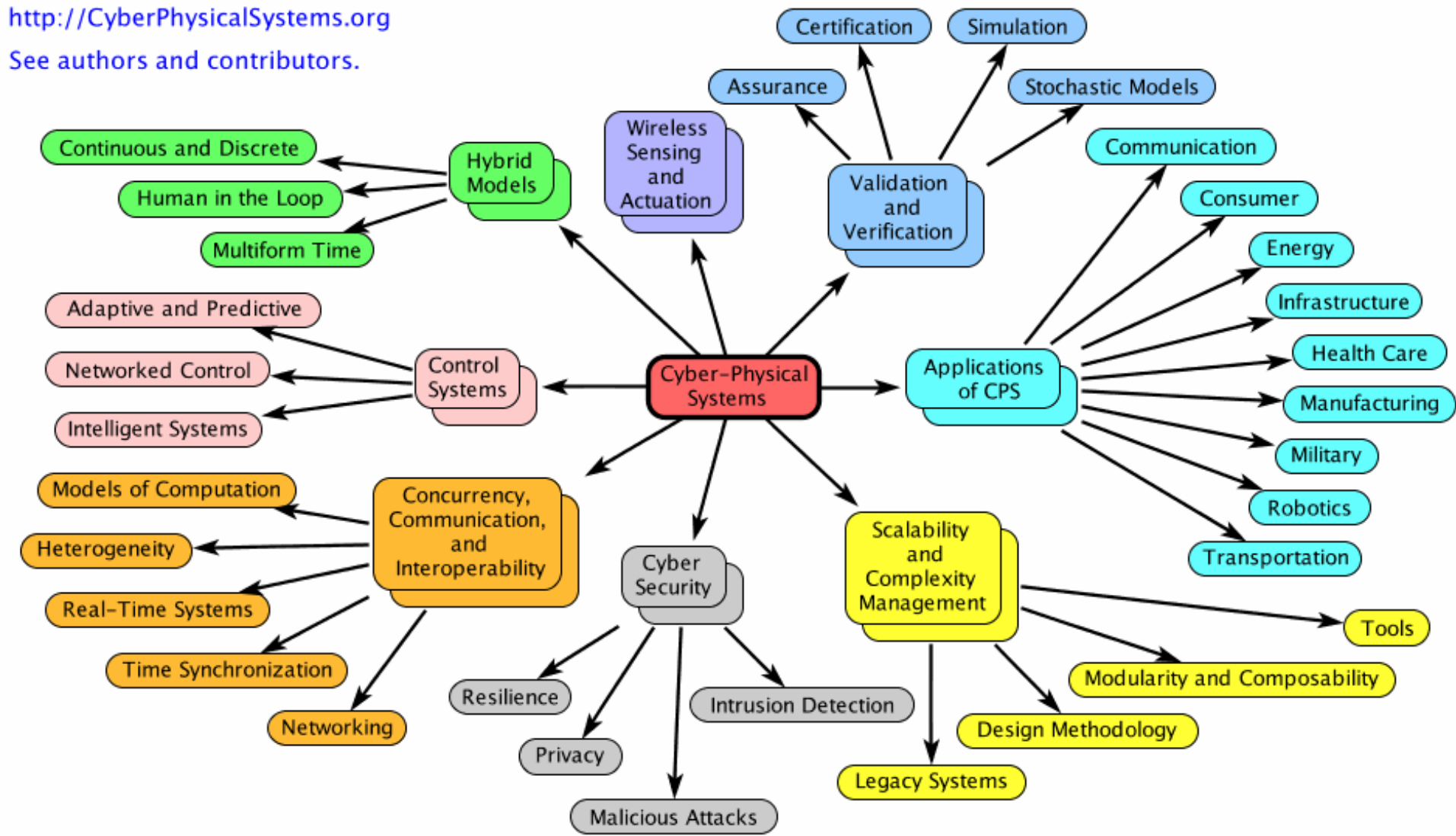
- Exemple classique : les systèmes **critiques**
  - Avions, automobiles, drones, centrales nucléaires, appareillage médical, etc.
- Mais aussi, plein de produits de consommation:
  - Multimedia (e.g. smartphones, appareil photo, etc.)
  - Matériel réseau (e.g. Box Internet)
  - Electroménager (robots domestiques, lave-linges...)
- Convergence forte entre ces domaines applicatifs
  - Voiture/avion autonome, connecté (contrôle classique + multimédia + IA + robotique...)

# Système de contrôle embarqué

## *Cyber-Physical Systems – a Concept Map*

<http://CyberPhysicalSystems.org>

See authors and contributors.



# 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, TensorFlow, 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

- Systèmes qui **réagissent** aux **stimuli** venant de l'environnement.
  - **Réactif vs. transformationnel** = question de point de vue (Harel&Pnueli, 1985)
  - 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

# Système réactif

- Les notions de stimulus et réponse doivent être vues en un sens très large:
  - On peut être forcé de faire des calculs pour déterminer quand un stimulus arrive (e.g. échantillonner des senseurs et faire des calculs pour déterminer qu'un avion est trop incliné).
  - Le temps peut être lui-même un stimulus, e.g. par l'utilisation d'alarmes.
  - Spécification en temps discret ou en temps continu.
- Difficile de trouver aujourd'hui des systèmes informatiques qui ne sont 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 (**concurrency**) 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.
    - 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.

# 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
  - Échéance d'une tâche : date après l'arrivée où la tâche doit être finie.
  - Types de tâches:
    - Périodiques – arrivent à des intervalles fixes
    - Sporadiques – arrivent avec une distance minimale entre elles
    - Apériodiques – peuvent arriver sans restrictions

# Ordonnancement temps-réel

- Allocation des ressources aux tâches dans le but d'assurer le respect des échéances
  - Déclenchement/interruption/reprise de tâches
  - Allocation mémoire...
- On peut le faire :
  - **En ligne** - lors de l'arrivée des stimuli, suivant des politiques (algorithmes) d'ordonnancement généralistes
  - **Hors ligne** - dates de départ choisies avant exécution, politique d'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)

```
for(;;) {
    x=read(fin,buf,0_NONBLOCK);
    if(x!=0) y=g(buf);
    else y=h();
    write(fout,y);
}
```

time	CPU0	CPU1	CPU2
0	read		
1			
2		if(x!=0) g	if(x=0) h
3			
4			
5			
6			write
7			
8			

# Ordonnancement **hors ligne**

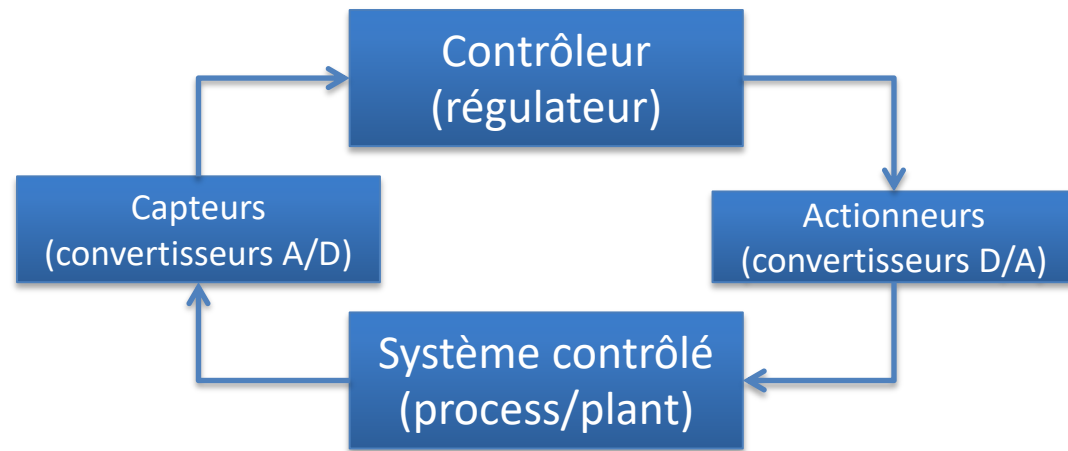
- Avantages:
  - Déterminisme temporel, **plus facile à vérifier/tester/certifier**
  - Pas besoin de marges temporelles (utilisation 100%)
  - Prise en compte facile des conditions d'exécution
- Désavantages:
  - Allocation statique des ressources, au pire cas (OK pour systèmes très critiques)
  - Vitesse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

# Programmation synchrone

- Langages pour les systèmes de contrôle-commande (embarqués)

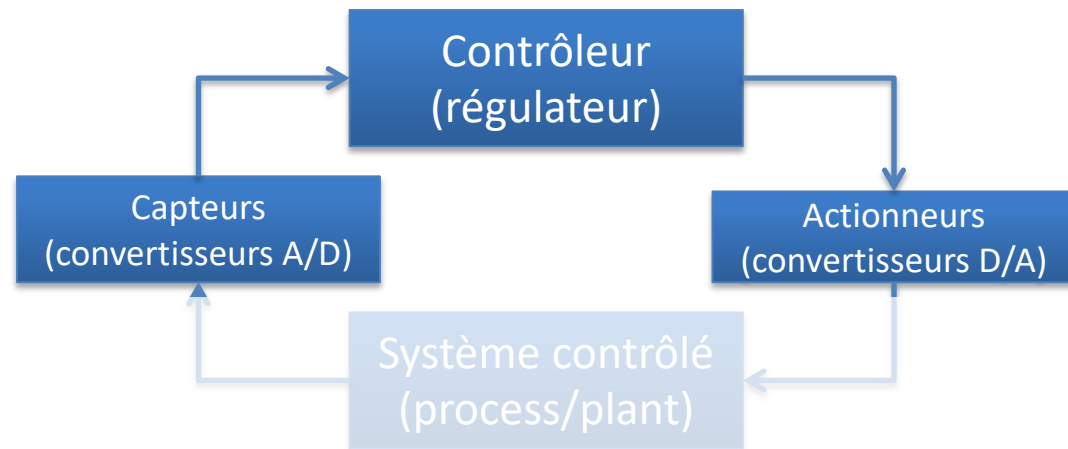


- Autres utilisations possible (signal processing, ML,...)
  - SP, ML souvent utilisés dans des systèmes embarqués



# Programmation synchrone

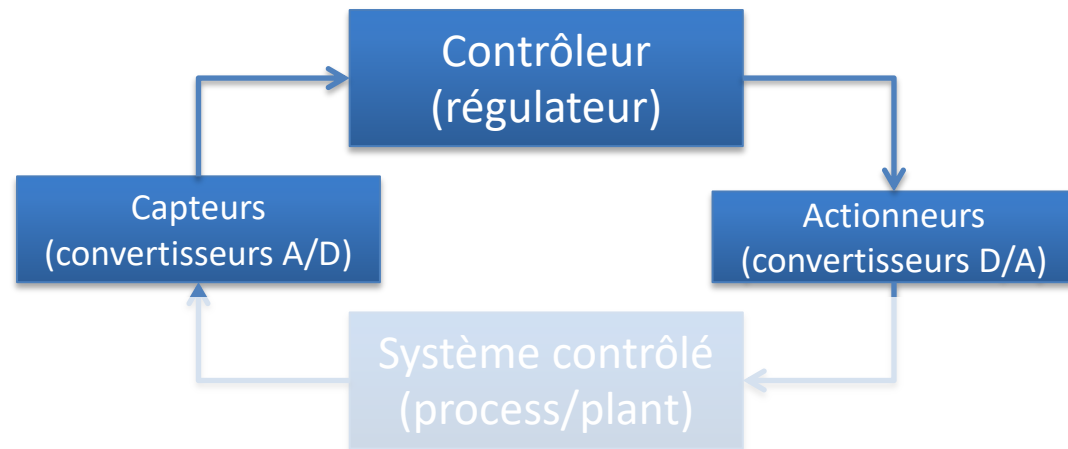
- Langages pour les systèmes de contrôle-commande (embarqués)



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

# Programmation synchrone

- Langages pour les systèmes de contrôle-commande (embarqués)



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

# Programmation synchrone

- Programmer ces systèmes est coûteux...
  - Coût très important : \$200-1000/LoC
    - <https://sel4.systems/Info/Docs/GD-NICTA-whitepaper.pdf>
    - [https://en.wikipedia.org/wiki/Evaluation\\_Assurance\\_Level#EAL6:\\_Semiformally\\_Verified\\_Design\\_and\\_Testing](https://en.wikipedia.org/wiki/Evaluation_Assurance_Level#EAL6:_Semiformally_Verified_Design_and_Testing)
- ... 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), etc.

(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

# Programmation synchrone

- Programmer ces systèmes est difficile :
  - Logiciels de très grande taille
    - Centaines de milliers de lignes de code, en rapide augmentation
  - 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
    - 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 en Heptagon

- Flot de données = style de présentation des algorithmes
  - Naturel en:
    - Conception de circuits digitaux synchrones (netlists)
    - Automatique (Simulink)
    - Systèmes embarqués critiques (Scade/Lustre)
    - Algorithmique multimédia/IA/Big Data (TensorFlow...)
  - 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 en Heptagon

- La brique de base: la fonction de calcul

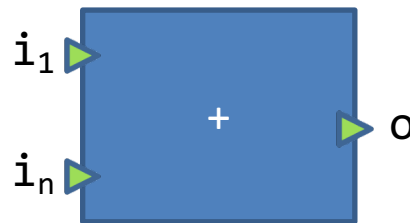


$$(o_1, \dots, o_m) = f(i_1, \dots, i_n)$$
$$o = f(i_1, \dots, i_n)$$

- 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

# Programmation synchrone en Heptagon

- Exemple :



$$o = i_1 + i_2$$

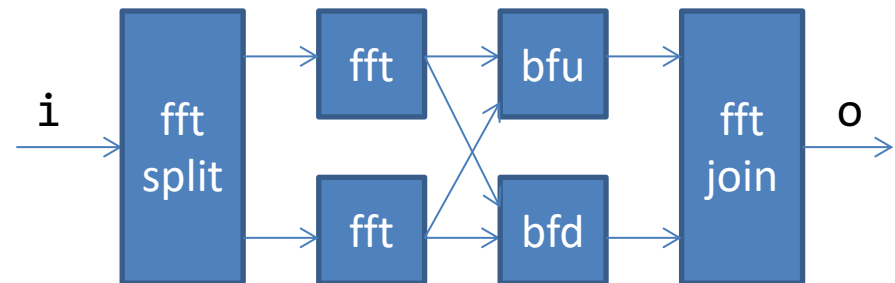
cycle	0	1	2	3	4	5	6	...
$i_1$	2	7	12	17		22	27	...
$i_2$	14	12	10	8		6	4	...
$o$	16	19	22	25		28	31	...

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

# Programmation synchrone en Heptagon

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

```
(i1,i2) = fft_split(i) ;  
v1 = fft(i1) ;  
v2 = fft(i2) ;  
o1 = bfu(v1,v2) ;  
o2 = bfd(v1,v2) ;  
o = fft_join(o1,o2) ;
```



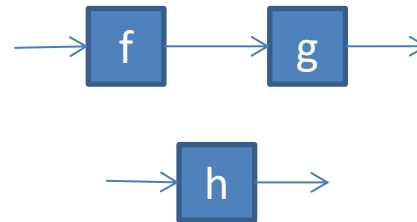
- Une fonction peut être appelée (instanciée) plusieurs fois
- Chaque port d'entrée est connecté à exactement un port de sortie (pour éviter les ambiguïtés)
- Mais un port de sortie peut être connecté à 0, 1, ou plusieurs ports d'entrée

# Programmation synchrone en Heptagon

- Pas de séquençement implicite
  - Dépendances de données explicites ➡ Concurrency

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

Ordre d'exécution fixé en C



Plusieurs ordres possibles  
en Heptagon :

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

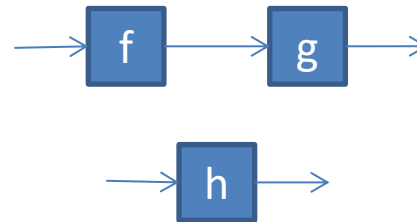
- Ordre partiel ➡ Tout ordre total le contenant est une exécution mono-processeur correcte.
  - Important : absence d'effets de bords des noeuds !

# Programmation synchrone en Heptagon

- Pas de séquençement implicite
  - Dépendances de données explicites ➡ Concurrency

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

Ordre d'exécution fixé en C



Plusieurs ordres 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.

– Rmq: Les compilos, e.g. GCC/LLVM, font de même: analyse de données, SSA.

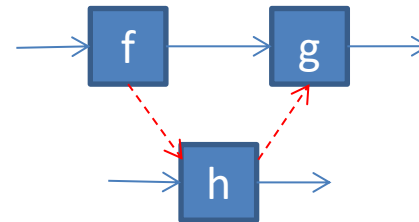
# Programmation synchrone en Heptagon

- Pas de séquençement implicite
  - Dépendances de données explicites ➡ Concurrency



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

Ordre d'exécution fixé en C



Plusieurs ordres possibles, ~~f;g;h;~~  
f;h;g;  
**mais on peut ajouter des** ~~h;f;g;~~  
**dépendances...**

- Ordre partiel ➡ Tout ordre total le contenant est une exécution mono-processeur correcte.

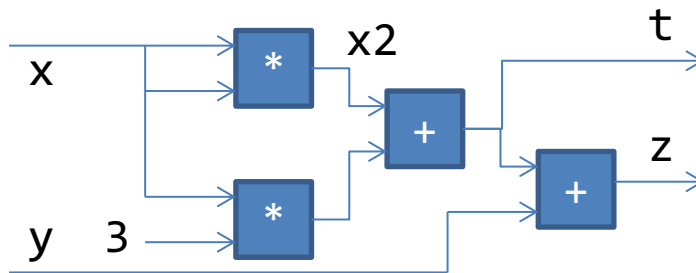
- Degré de liberté pour la génération de code efficace.

– Rmq: Les compilos, e.g. GCC, font de même: analyse de données, SSA.

# Programmation synchrone en Heptagon

- Expressions

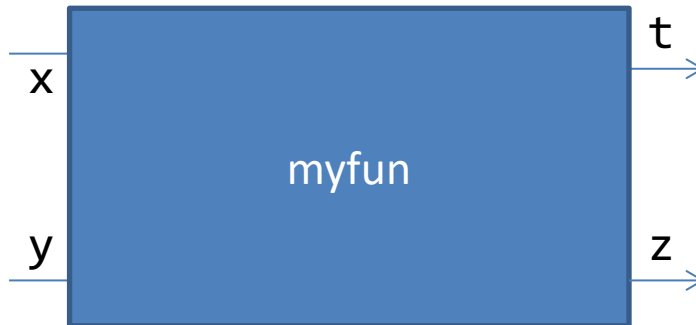
```
x2 = x*x ;  
t = x2 + 3*x ;  
z = t + y ;
```



# Programmation synchrone en 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
```



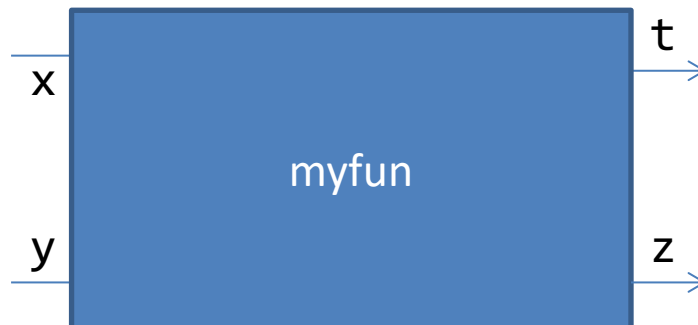


# Programmation synchrone en Heptagon

- 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) ;
...
```

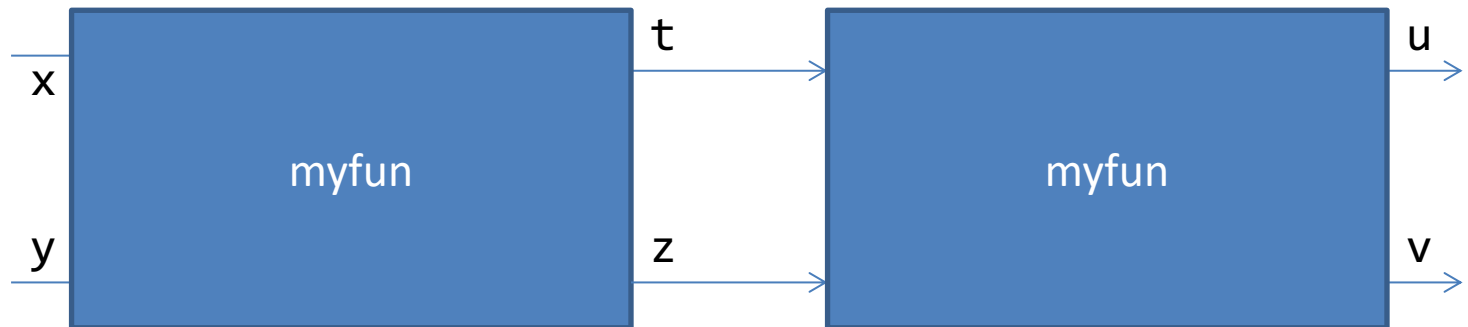


# Programmation synchrone en Heptagon

- 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) ;
...
```



# Programmation synchrone en Heptagon

- Typage strict des données
  - Pas de polymorphisme, même pour l'arithmétique
    - $1 + 3 : \text{int}$
    - $1. +. 3. : \text{float}$
  - Types de données élémentaires :
    - `bool` : `true`, `false`
    - `int` : opérateurs `+`, `-`, `*`, `/`
    - `float` : opérateurs `+`, `-`, `*`, `/`
    - `string` : implemented by `char*`

# Programmation synchrone en Heptagon

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

# Programmation synchrone en Heptagon

- État d'un programme – instruction fby

$z = x \text{ fby } y ;$

$$z_0 = x_0$$

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

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	...

# Programmation synchrone en Heptagon

- État d'un programme – instruction fby

$z = x \text{ fby } y ;$

$$z_0 = x_0$$

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

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

# Programmation synchrone en 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	...

# Programmation synchrone en Heptagon

- Un compteur

```
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	...

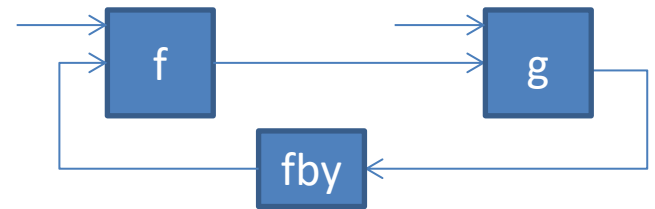
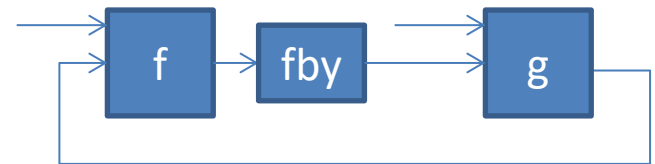


# Programmation synchrone en Heptagon

- Causalité :
  - Tout cycle du graphe flot de données doit contenir un fby



Version incorrecte



Versions correctes minimales

- Règle assurant le déterminisme
  - Sinon : qui s'exécute en premier, et sur quelle donnée ?

# Programmation synchrone en Heptagon

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

# Programmation synchrone en 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
```

# Programmation synchrone en 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

# Programmation synchrone en 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;
```

# Programmation synchrone en 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**

# Programmation synchrone en Heptagon

- Compilation de `first.ept`

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

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

```
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;
```

# Programmation synchrone en Heptagon

- 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 */  
    }  
}
```



# Programmation synchrone en Heptagon

- 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

# Programmation synchrone en Heptagon

- 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>
      - Si "opam install heptagon" échoue, exécuter "sudo apt-get install camlp4", puis retenter "opam install heptagon"
      - La version d'Ubuntu et le type d'installation comptent
    - Documentation sur le langage : <http://heptagon.gforge.inria.fr/pub/heptagon-manual.pdf>
      - Posez-moi des questions !
  - Écriture des premiers programmes synchrones
    - Compilation
    - Exécution du code

# Préparation du TP

- Exemples à programmer (transparent 52) :
  - Objectif 1 : myfun
  - Objectif 2 : sum2
  - Objectif 2 : counter
- Il faut pour chaque exemple :
  - Le programmer
  - 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 XXX, créer un répertoire XXX-NOM (ou NOM est le nom de l'étudiant)
  - Placer dedans le code Heptagon, le code C écrit à la main (main.c) et le script de compilation (ou Makefile)
- 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 mettre du code C écrit manuellement dedans
  - Ne jamais modifier un fichier C généré (e.g. pour y logger la fonction main)

# Préparation du TP

- Comment rendre un objectif fini :
  - M'appeler, et me le montrer sur ordinateur
  - Ensuite archiver le répertoire XXX-NOM en format .tar.gz ou .zip, et m'envoyer l'archive par mél.