

# Conception de systèmes embarqués temps réel

Dumitru Potop-Butucaru

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

cours EPITA, 2025

# Contenu

- Introduction au « temps réel », 2<sup>ème</sup> partie
- Spécification fonctionnelle synchrone en Heptagon
  - Notions fondamentales
    - Temps logique
    - Horloge logique
- Préparation du TP

# 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 de contrôle embarqué

- Exigences non-fonctionnelles complexes au niveau système :

- Temps réel

- Efficacité
- Prédictabilité



- Basse consommation

- Green computing



- Sûreté de fonctionnement, sécurité

- Eviter accidents et actions malveillantes
- Isolation des applications
- Tolérance aux pannes
- Confidentialité



- Coût (argent/temps/...)

- Plate-forme/développement/exploitation



- Flexibilité

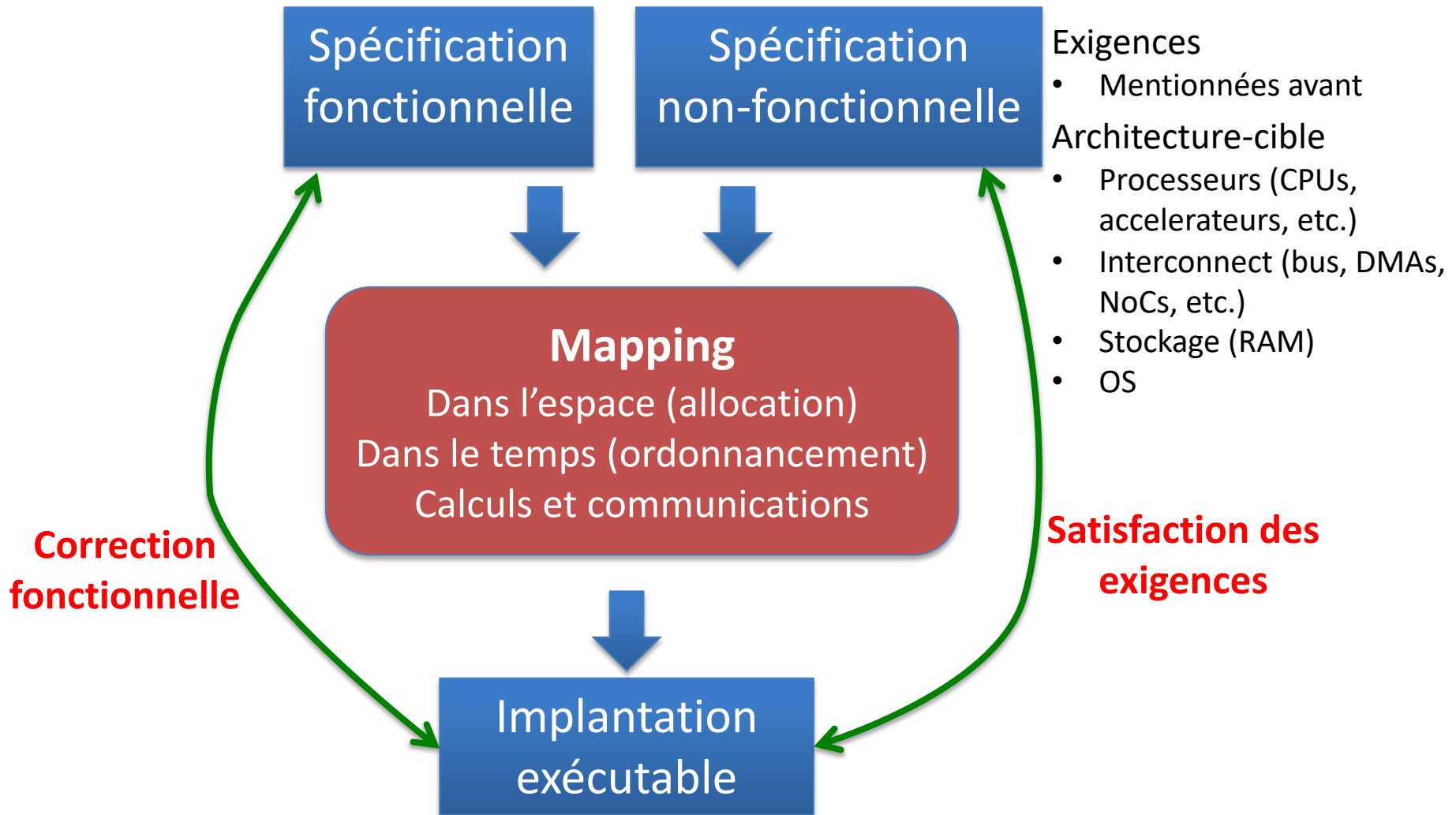
- Evolution du système

- Taille

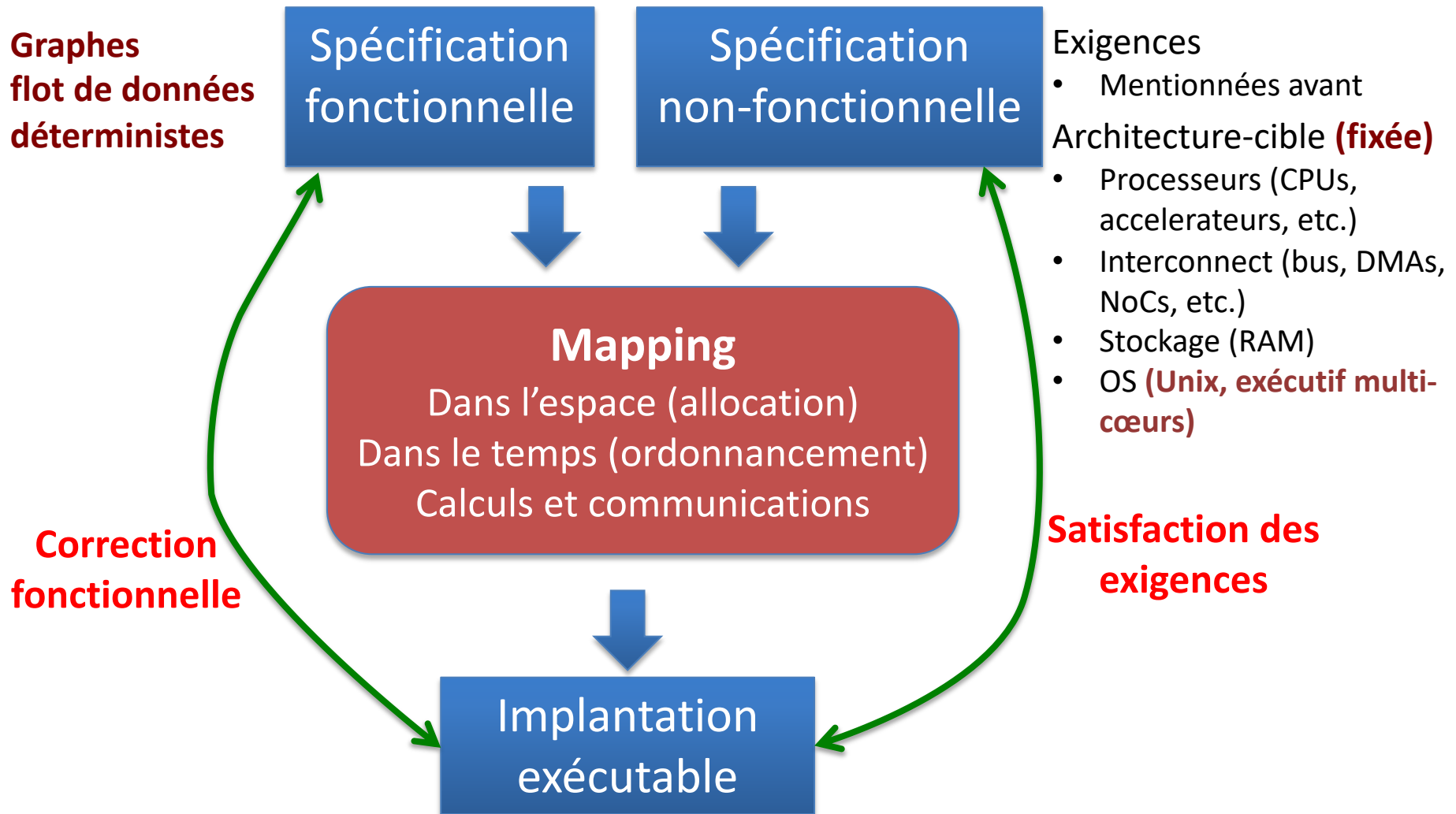
- Dissipation thermique

- ...

# Implantation des systèmes embarqués



# Implantation des systèmes embarqués

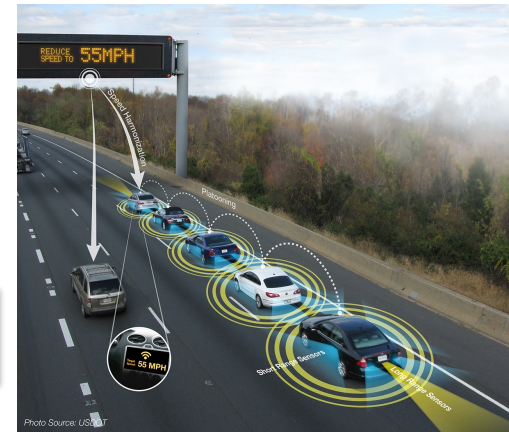
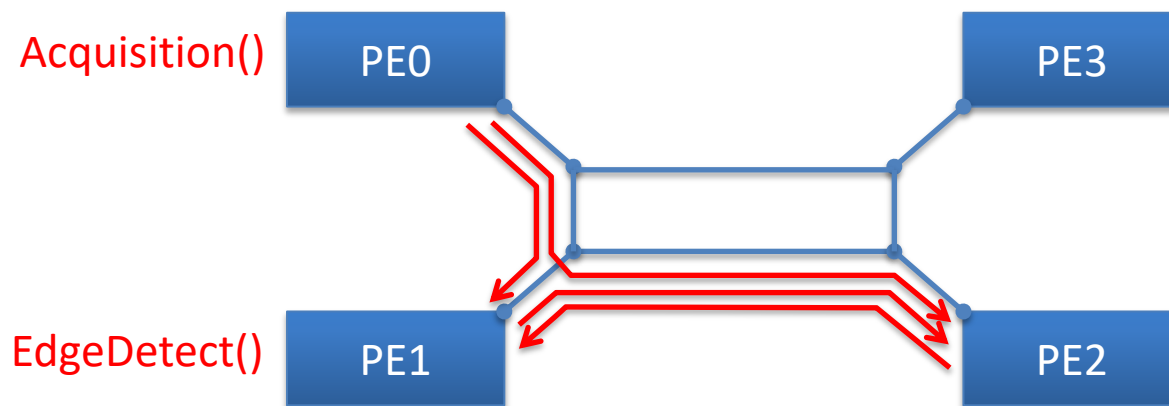


# Implantation des systèmes embarqués

- Mapping dans l'espace
  - Où (**par qui**) l'opération est réalisée ?
  - Vocabulaire :
    - CPU/RAM: **allocation, distribution**
    - Interconnect: **routage**

# Implantation des systèmes embarqués

- Mapping dans l'espace
  - Où (par qui) l'opération est réalisée ?
  - Vocabulaire :
    - CPU/RAM: **allocation, distribution**
    - Interconnect: **routage**

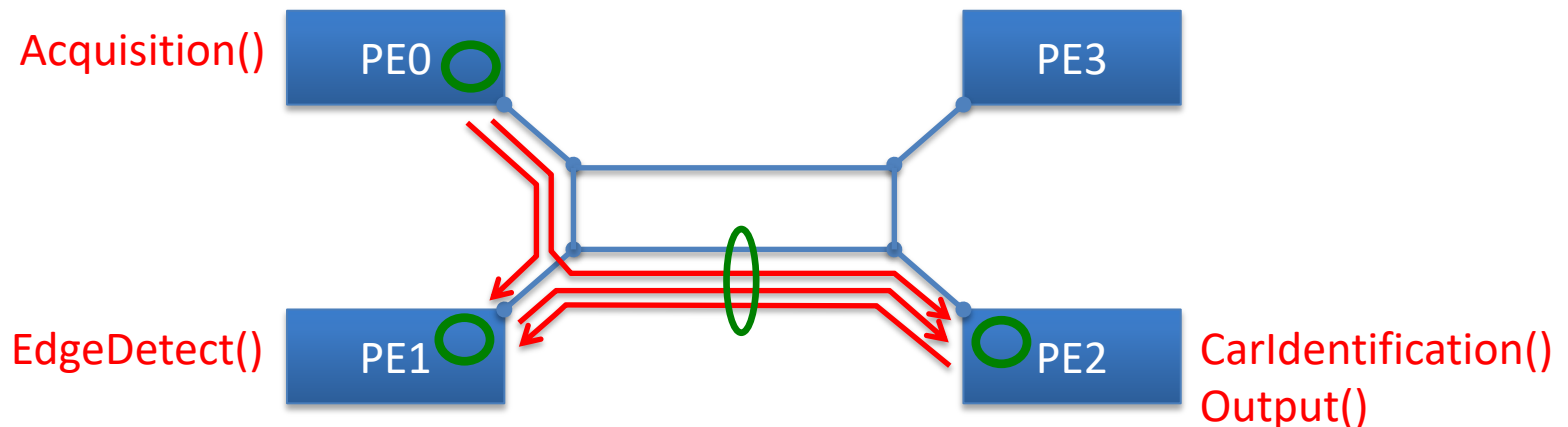


CarIdentification()  
Output()



# Implantation des systèmes embarqués

- Mapping dans le temps
  - **Quand** l'opération est réalisée (à quelle date, dans quel ordre)
    - Allocation des ressources dans les systèmes concurrents
  - Vocabulaire:
    - CPU: **ordonnancement, scheduling, séquencage**
    - Interconnect: **arbitration, ordonnancement, séquencage**



# Implantation des systèmes embarqués

- Complexité des algorithmes de mapping
  - Optimal: NP-dur dans les meilleurs des cas, impossible en pratique
  - Heuristiques (techniques fondées sur l'expérience)
    - Parmi elles :
      - Les politiques classiques d'ordonnancement (RM, EDF, etc.)
      - Les algorithmes de compilation (gcc, llvm...)
    - Une heuristique peut être optimale ou formellement caractérisée sous certaines hypothèses restrictives (e.g. EDF optimal en mono-processeur avec coût de préemption négligé)
- Classification des techniques de mapping. Critère 1 (déjà introduit):
  - **Hors ligne/Statique**
    - Décisions prises avant l'exécution
      - L'exécution conditionnelle peut être prise en compte plus facilement
    - Aucune imprécision temps/ordre dans un certain référentiel (après abstraction)
  - **En ligne/Dynamique**
    - Il reste de l'imprécision en temps/ordre.
    - Les décisions de mapping dépendent d'aspects du système qui n'ont pas été spécifiés ou analysés hors ligne (trop complexe), ou l'ordonnancement statique est inapplicable (tables trop grandes) :
      - Dates d'arrivée des événements
      - Variations du temps d'exécution
      - Aspects internes/inobservables de l'OS/HW/etc.

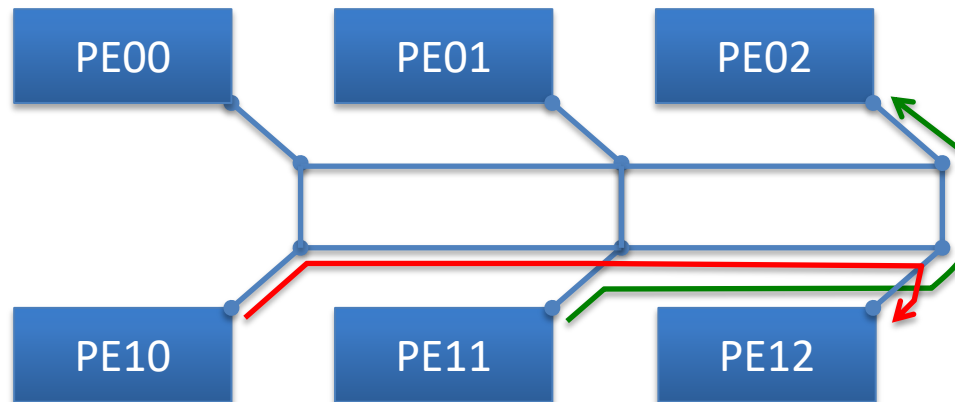
# Implantation des systèmes embarqués

- Exemple: routage sur un réseau sur puce :
  - Problème: transmettre 2 données entre deux coeurs



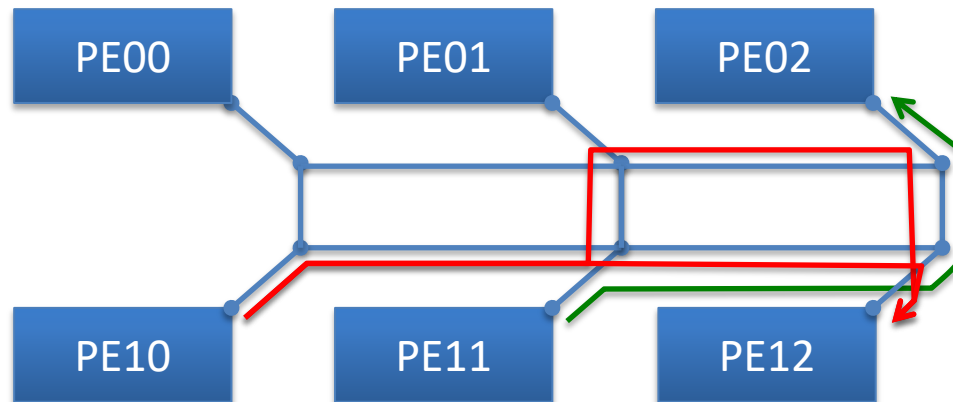
# Implantation des systèmes embarqués

- Exemple: routage sur un réseau sur puce :
  - Problème: transmettre 2 données
  - Routage statique (X-first):



# Implantation des systèmes embarqués

- Exemple: routage sur un réseau sur puce :
  - Problème: transmettre 2 données
  - Routage dynamique (adaptatif) :



# Implantation des systèmes embarqués

- Exemple: ordonnancement sur un CPU :

- Problème: exécuter cycliquement f() et g()

- Ordonnancement dynamique :

**Process1:**

```
for(;;){  
    f();  
}
```

**Process2:**

```
for(;;){  
    g();  
}
```

Lancer les processus sous Linux


- Ordonnancement statique :

```
for(;;){  
    f();  
    g();  
}
```

Contraintes :

- Périodes égales
- Dépendances à respecter

# Ordonnancement/arbitration (classification)

- Critère 2 : complexité de l'algorithme (politique) d'ordonnancement
    - Plus simple: Ordre fixe, FIFO
    - Politiques équitables
      - (weighted) round robin
      - (weighted) fair queuing, etc.
    - À base de priorités
      - Priorités statiques : FP, RM, DM
      - Priorités dynamiques : EDF, LLF
      - ...
    - Algorithmes hors ligne (heuristiques ou « exacts »)
      - Décisions prises hors ligne sont appliquées en ligne
- 
- Applicables en ligne  
(basse complexité)

# Ordonnancement/arbitration (classification)

- Critère 2 : complexité de l'algorithme (politique) d'ordonnancement
  - Plus simple: Ordre fixe, FIFO
  - Politiques équitables
    - (weighted) round robin
    - (weighted) fair queuing, etc.
  - À base de priorités
    - Priorités statiques : FP, RM, DM
    - Priorités dynamiques : EDF, LLF
    - ...
  - Algorithmes hors ligne (heuristiques ou « exacts »)
    - Décisions prises hors ligne sont appliquées en ligne
- Autres critères :
  - Event-driven vs. Time-triggered (comment on déclenche l'exécution ?)
  - Preemptif vs. Non-preemptif (peut-on interrompre une opération ?)
  - Criticalité simple vs. Criticalité mixte (importance d'une tâche ?)
  - Tolérant aux pannes ou non ?
  - Ordonnancement partitionné vs. global, etc.

Applicables en ligne  
(basse complexité)



# Ordonnancement multiprocesseur

- **Mapping du système =  
Mappings sur CPUs + mapping sur l'interconnect**
  - Les points limitants peuvent être dans les CPU, l'interconnect, ou une combinaison des deux
    - Dépend du matériel, de l'application, et du mapping lui-même
    - computation-intensive vs. communication-intensive
  - **Divers algorithmes sont nécessaires dans différents contextes embarqués (sur CPU et interconnect)**
    - L'ordonnancement FIFO est simple/peu cher
    - Les algorithmes équitables sont utilisés en temps réel mou (e.g. traitement de signal), et certains permettent l'analyse modulaire.
    - L'ordonnancement à base de priorités est utile pour assurer des temps de réponse courts pour quelques tâches importantes
    - L'ordonnancement hors ligne est utile pour les traitements périodiques (contrôle discrétisé) et pour assurer une meilleure prédictibilité dans les systèmes critiques ...

# Programmation en langage synchrone Heptagon

2ème partie

# Avantages du synchrone flot de données

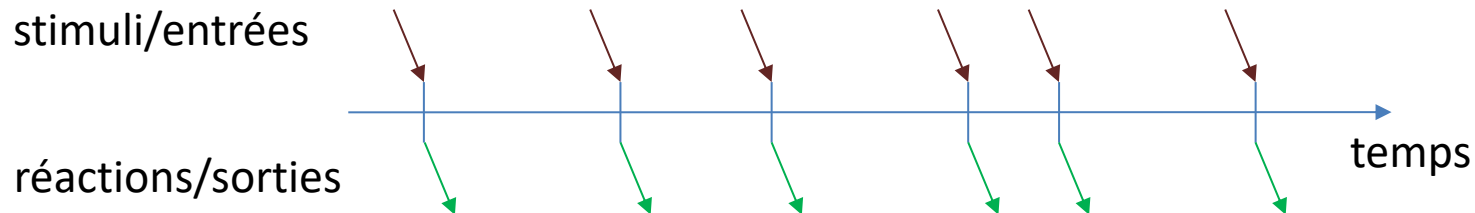
- Proche de **Simulink** (et VHDL/Verilog, Tensorflow/Pytorch...)
  - Standard de facto pour la description de systèmes de contrôle-commande
- **Correction par construction**
  - Sans effets de bords
  - Sans variables non-initialisées
  - Concurrence déterministe
  - Sans comportements infinis (sauf au niveau système)
- Test/analyse/debug/preuve/**certification** facilités
  - Déterminisme fonctionnel – un seul comportement pour un jeu de données d'entrée, même en présence de multi-tâches/distribution
  - Modèle synchrone = moins d'états à explorer
- **Synthèse** d'implantations facilitée
  - Proche des modèles de tâches du temps réel
  - Ordonnancement, génération de code

# Que peut-on programmer avec ?

- Tout ce que l'on peut programmer en C **sans appels de fonctions à récursion non-bornée.**
  - Si on n'utilise pas des types infinis => **automates finis, circuits digitaux synchrones**
- Autres langages synchrones
  - Flot de données (**SCADE**, Lustre, Signal, SynDEx)
  - Flot de contrôle (Esterel, Quartz)
- Apparentées: **Simulink/Stateflow, VHDL, Keras...**
  - Simulink/VHDL sont synchrones sous certaines hypothèses (sous-ensembles sûrs ou synthétisables)

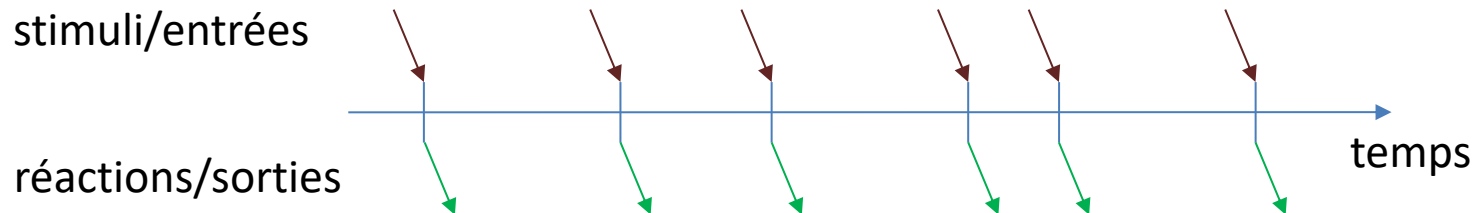
# Temps logique discret

- Système réactif = **réagit aux stimuli** venant de l'environnement
  - Hypothèse supplémentaire – interaction en **temps discret**
    - Les stimuli sont pris en compte en une **séquence de cycles d'exécution**
    - L'acquisition d'entrées et la production de réactions se font seulement aux frontières entre cycles (**les calculs sont atomiques**)
      - Facilite la définition d'une sémantique concurrente et déterministe

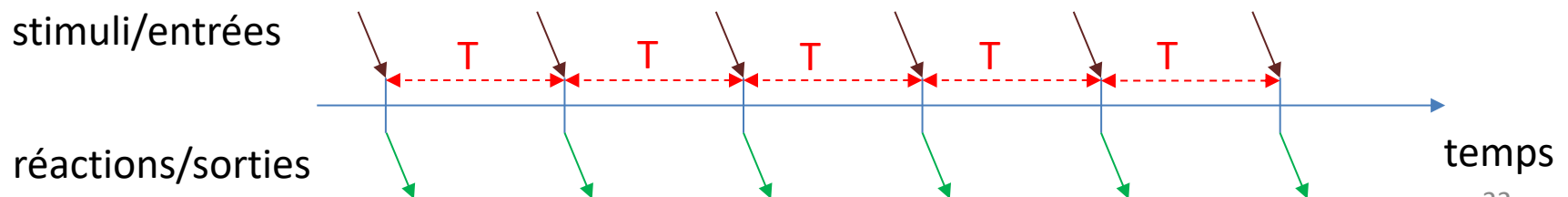


# Temps logique discret

- Système réactif = **réagit aux stimuli** venant de l'environnement
  - Hypothèse supplémentaire – interaction en **temps discret**
    - Les stimuli sont pris en compte en une **séquence de cycles d'exécution**
    - L'acquisition d'entrées et la production de réactions se font seulement aux frontières entre cycles (**les calculs sont atomiques**)
      - Facilite la définition d'une sémantique concurrente et déterministe

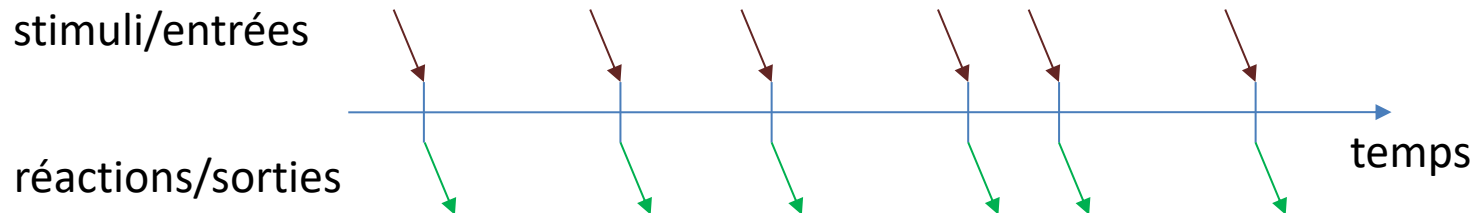


- Facile à lier au temps réel "physique" classique (e.g. périodique)

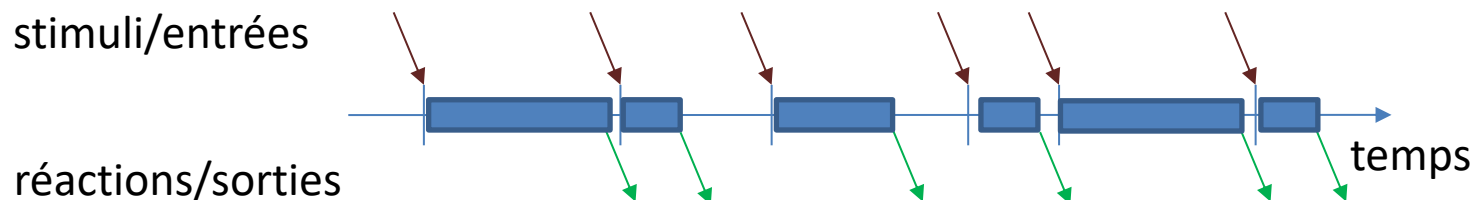


# Temps logique synchrone

- Synchrone = paradigme en temps logique
  - Modèle formel – calculs instantanés
    - Sorties et entrées sont synchrones -> définition formelle naturelle de la composition (produit synchrone)

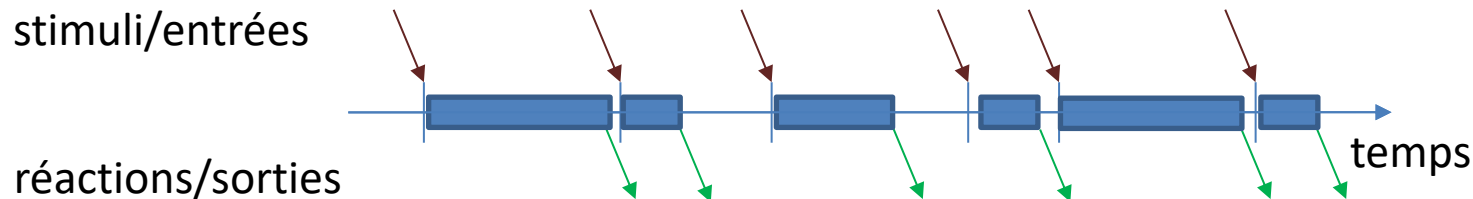


- Implantation temps réel – un calcul démarré dans un cycle doit finir avant le prochain cycle



# Temps logique synchrone

- Synchrones = paradigme en temps logique
  - Modèle formel – calculs instantanés
    - Sorties et entrées sont synchrones -> définition formelle naturelle de la composition (produit synchrone)
  - Implantation temps réel – un calcul démarré dans un cycle doit finir avant la fin du cycle



```
loop
  await_stimulus() ;
  read_inputs() ;
  compute() ;
  write_outputs() ;
end
```

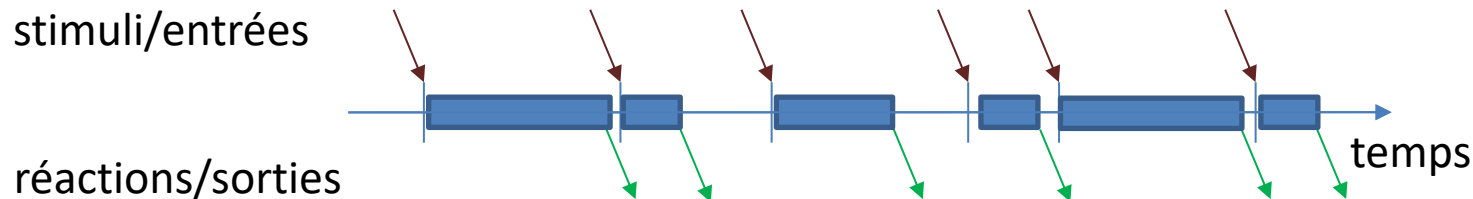
Qu'est-ce qu'un trigger?

- Toutes les entrées présentes
- Timer périodique, avec échantillonnage des entrées
- Au moins une entrée présente, et les autres gardent l'ancienne valeur



# Temps logique synchrone

- Adapté à plusieurs domaines :
  - Contrôle-commande classique
  - Multimédia
  - Algorithmique Machine Learning (réseaux de neurones)
  - Modélisation de circuits digitaux



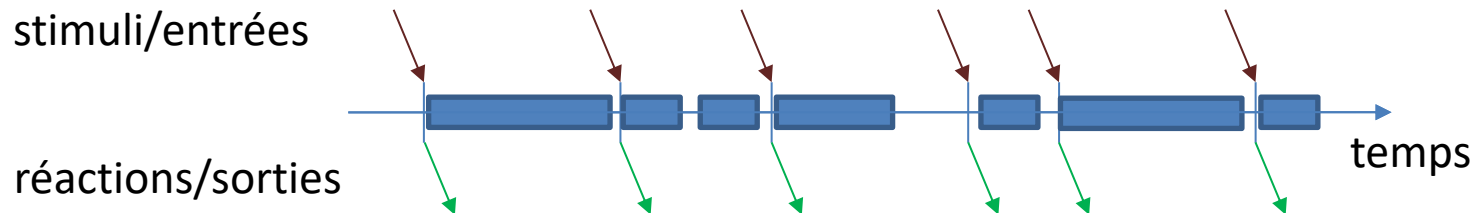
```
loop
  await_stimulus() ;
  read_inputs() ;
  compute() ;
  write_outputs() ;
end
```

Qu'est-ce qu'un trigger?

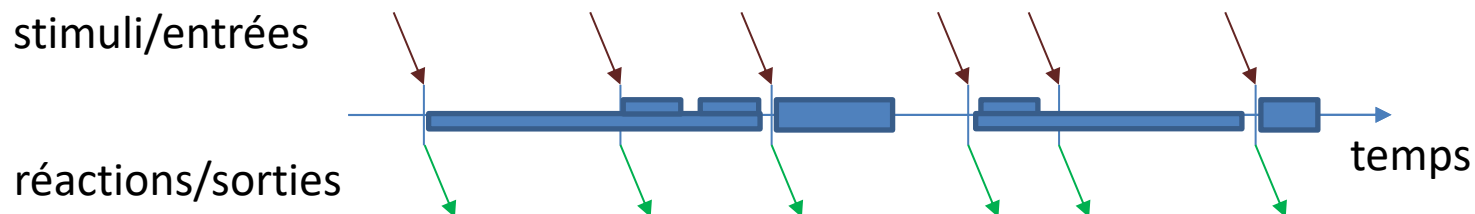
- Toutes les entrées présentes
- Timer périodique, avec échantillonnage des entrées
- Au moins une entrée présente, et les autres gardent l'ancienne valeur

# Les limites du synchrone

- LET = logical execution time
  - I/O se font sur les tops de l'horloge logique
  - Chaque calcul a une durée en temps logique



- Un calcul peut s'étendre sur plus d'un cycle



# Programmation synchrone en Heptagon

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

# Programmation synchrone en Heptagon

- Contrôle simple – instruction if

`z = if c then x else y`



Toutes les valeurs sont présentes aux mêmes cycles. Ce n'est pas de l'exécution conditionnelle, mais un choix conditionnel.

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

# Programmation synchrone en Heptagon

- Un compteur **avec reset**

```
node rcounter(rst:bool) returns (cnt:int)
let
  cnt = if rst then 0 else (0 fby (cnt+1))
tel
```

cycle	0	1	2	3	4	5	6	7	...
rst	f	f	f		f	t	t	f	...
cnt	0	1	2		3	0	0	1	...

# Programmation synchrone en Heptagon

- Ce que l'on a défini déjà :
  - Appel de fonctions
  - Hiérarchie
  - État
  - Contrôle simple (instruction if)
- Beaucoup de systèmes industriels en avionique, rail, automobile, peuvent être programmés ainsi (et le sont !)
  - Tout exécuter à l'avantage de la prédictibilité temporelle

# Programmation synchrone en Heptagon

- Ce qui manque : exécution conditionnelle

```
if(c) {  
    f(...) ;  
} else {  
    g(...) ;  
}
```

- Nécessaire pour :

- Exécution efficace – réduire l'utilisation des ressources
  - E.g. multi-périodes, modes fonctionnels
- Contrôle conditionnel des actionneurs
  - Peut être encapsulé dans du code qui est toujours exécuté

# Programmation synchrone en Heptagon

- Notion fondamentale : l'horloge logique
  - Condition d'activation
    - Décrit la suite de cycles d'exécution (du temps logique) où cette condition est vraie
  - Chaque variable ou calcul a une horloge
    - Horloge d'une variable: horloge logique définissant quand le signal est présent avec une valeur
      - Le signal ne peut pas être utilisé dans les calculs aux autres cycles
      - Mais il doit être bien initialisé aux cycles de son horloge
    - Horloge d'un calcul: horloge logique définissant quand un calcul est réalisé



# Horloges logiques

- Dans une trace, on peut énumérer ces cycles
  - Horloge de la variable x : {0,1,2,4,6...}
  - Horloge de la variable z : {2,6...}

cycle	0	1	2	3	4	5	6	...
x	10	7	21		9		33	...
c	t	t	f		t		f	...
y	10	7			9			...
z			21				33	...

# Horloges logiques

- Avant l'exécution, on peut les identifier par des "expressions d'horloges" (des prédicats)
  - $\text{Clk}(y)$  = quand  $c$  est présent et vrai
  - $\text{Clk}(z)$  = quand  $c$  est présent et faux

cycle	0	1	2	3	4	5	6	...
x	10	7	21		9		33	...
c	t	t	f		t		f	...
y	10	7			9			...
z			21				33	...

$$c = (x < 11)$$

# Définir une horloge (1/4)

- Sous-échantillonnage

`y = x when c ;`

`z = x whenot c ; (* pareil que « x when (not c) » *)`

cycle	0	1	2	3	4	5	6	...
x	10	7	21		9		33	...
c	t	t	f		t		f	...
y	10	7			9			...
z			21				33	...

`c = (x < 11)`

# Définir une horloge (2/4)

- Horloge héritée par le flot de données

`y = x when c ;`

`z = x whenot c ; (* pareil que « x when (not c) » *)`

`t = f(y) ; (* exécuté quand y présent *)`

cycle	0	1	2	3	4	5	6	...
x	10	7	21		9		33	...
c	t	t	f		t		f	...
y	10	7			9			...
z			21				33	...
t	f(10)	f(7)			f(9)			...

`c = (x < 11)`

# Définir une horloge (3/4)

- Reconstruction à partir de résultats partiels

```
y = x when c ;  
z = x whenot c ; (* pareil que « x when (not c) » *)  
  
t = f(y) ; (* exécuté quand y présent *)  
u = g(z) ; (* exécuté quand z présent *)  
r = merge c (true -> t) (false -> u) ; (* pareil que x *)
```

cycle	0	1	2	3	4	5	6	...
x	10	7	21		9		33	...
c	t	t	f		t		f	...
y	10	7			9			...
z			21				33	...
t	f(10)	f(7)			f(9)			...
r	f(10)	f(7)	g(21)		f(9)		g(33)	...

$c = (x < 11)$

# Définir une horloge (4/4)

- Chaque nœud a une horloge de base (tick = .)
  - Toutes les autres horloges du noeud en sont dérivées
    - when ou whenot appliqué zéro ou plusieurs fois
  - Toutes les I/O ont l'horloge « . » (dans ce cours)

# Horloge de base

- Chaque nœud a une horloge de base (tick = .)
  - Toutes les autres horloges du noeud en sont dérivées
    - when ou whenot appliqué zéro ou plusieurs fois
  - Toutes les I/O ont l'horloge « . » (dans ce cours)

```
node mynode(i:int;c:bool) returns (o:int) (* i, c, o ont horloge . *)
var i1,i2,o1,o2:int ; (* variables locales *)
let
  i1 = i when c ;    (* . when c *)
  i2 = i whenot c ;  (* . whenot c *)
  o1 = i1 + 1 ;      (* . when c *)
  o2 = i2 - 1 ;      (* . whenot c *)
  o = merge c (true->o1) (false->o2) (* . *)
tel
```

cycle	0	1	2	3	4	...
i	10	7	21	33	9	...
c	t	f	t	t	f	...
o	11	6	22	34	8	...

# Horloge de base

- Chaque nœud a une horloge de base (tick = .)
  - Toutes les autres horloges du noeud en sont dérivées
    - when ou whenot appliqué zéro ou plusieurs fois
  - Toutes les I/O ont l'horloge « . » (dans ce cours)

```
node mynode(i:int;c:bool) returns (o:int)
```

```
let
```

```
  o = merge c
```

```
    (true -> (i when c)+1)
```

```
    (false -> (i whenot c)-1);
```

```
tel
```

cycle	0	1	2	3	4	...
i	10	7	21	33	9	...
c	t	f	t	t	f	...
o	11	6	22	34	8	...



# Horloge de base

- Une horloge est toujours relative à un noeud
  - Le tick du nœud instancié est égal à l'horloge des variables d'entrée-sortie
  - Noeud racine: aucun sous-échantillonnage

```

node mynode(i:int;c:bool) returns (o:int)
let
    o = merge c
        (true -> (i when c)+1)
        (false -> (i whennot c)-1);
tel
node main(m:int) returns (n:int)
var d : bool ;
let
    d = m>0 ;
    n = merge d
        (true -> mynode(m when d, (m>9) when d))
        (false -> 11) ; tel
    
```

cycle	0	1	2	3	4	5	...
i = m when (m>0)	10	7	21	33		9	...
c = (m>9) when (m>0)	t	f	t	t		f	...
o	11	6	22	34		8	...
m	10	7	21	33	-2	9	...
n	11	6	22	34	11	8	

# Syntaxe plus familière

- Pseudo-impérative

```
node abc(i:int) returns (o:int)
let
  if i>0 then
    var a : int ; in
      a = i*i ;
      o = i+a ;
  else
    o = i + 1 ;
  end
tel
```

# Syntaxe plus familière

- Pseudo-impérative

```
node abc(i:int) returns (o:int)
let
  if i>0 then
    var a : int ; in (* var locales du bloc d'instructions *)
      a = i*i ;
      o = i+a ;
  else
    o = i + 1 ;
  end
tel
```

- Les variables non-locales doivent être affectées dans les deux branches du « if »

# Syntaxe plus familière

- Pseudo-impérative (**mais ce n'est pas du C**) :

```
node abc(i:int) returns (o:int)
```

```
let
```

```
  if i>0 then
```

```
    var a : int ; in
```

```
      a = i*i ;
```

```
      o = i+a ;
```

```
  else
```

```
    o = i + 1 ;
```

```
  end
```

```
tel
```

```
node abc1(i:int) returns (o:int)
```

```
var c : bool ;
```

```
  a,i1,i2 : int ;
```

```
let
```

```
  c = (i>0); i1 = i when c; i2 = i whennot c;
```

```
  a  = i1*i1 ;
```

```
  o1 = i1 + a ;
```

```
  o2 = i2 + 1 ;
```

```
  o = merge c (true -> o1) (false -> o2) ;
```

```
tel
```

# Syntaxe plus familière

- Pseudo-impérative (**mais ce n'est pas du C**) :

```
node jkl(i:int) returns ()  
let  
  if i>0 then  
    () = f() ;  
  else  
    () = g() ;  
  end  
tel
```

```
node jkl1(i:int) returns ()  
let  
  () = f() ;  
  () = g() ;  
tel
```

- Le conditionnement passe toujours par les vars
  - Le seul moyen de conditionner un appel de fonction sans entrées ou sorties -> l'incorporer à un nœud avec I/O

# Syntaxe plus familière

- Pseudo-impérative (**mais ce n'est pas du C**) :

```
node jkl2(i:int) returns ()  
let  
  if i>0 then  
    () = callf(i) ;  
  else  
    () = callg(i) ;  
  end  
tel
```

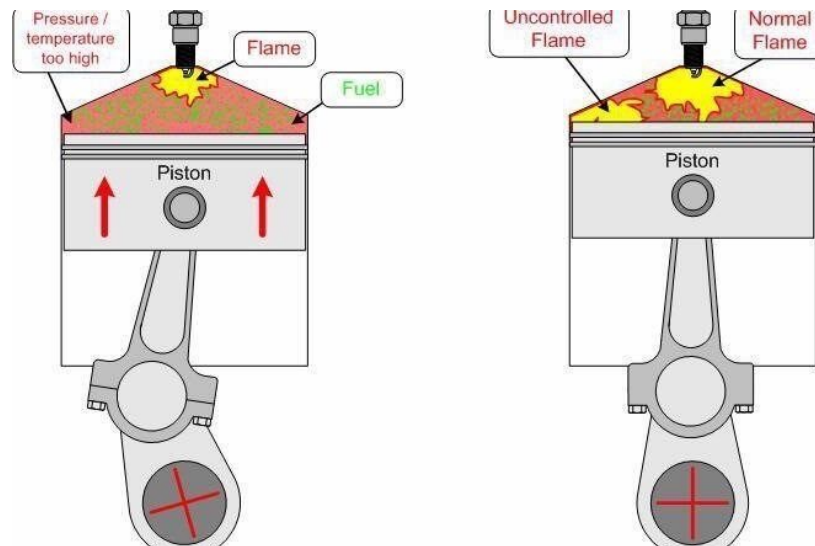
```
node callf(i:int) returns ()  
let  
  () = f() ;  
tel
```

```
node jkl3(i:int) returns ()  
let  
  () = callf(i when (i>0)) ;  
  () = callg(i whennot (i>0)) ;  
tel
```

- Le conditionnement passe toujours par les vars
  - Le seul moyen de conditionner un appel de fonction sans entrées ou sorties -> l'incorporer à un nœud avec I/O

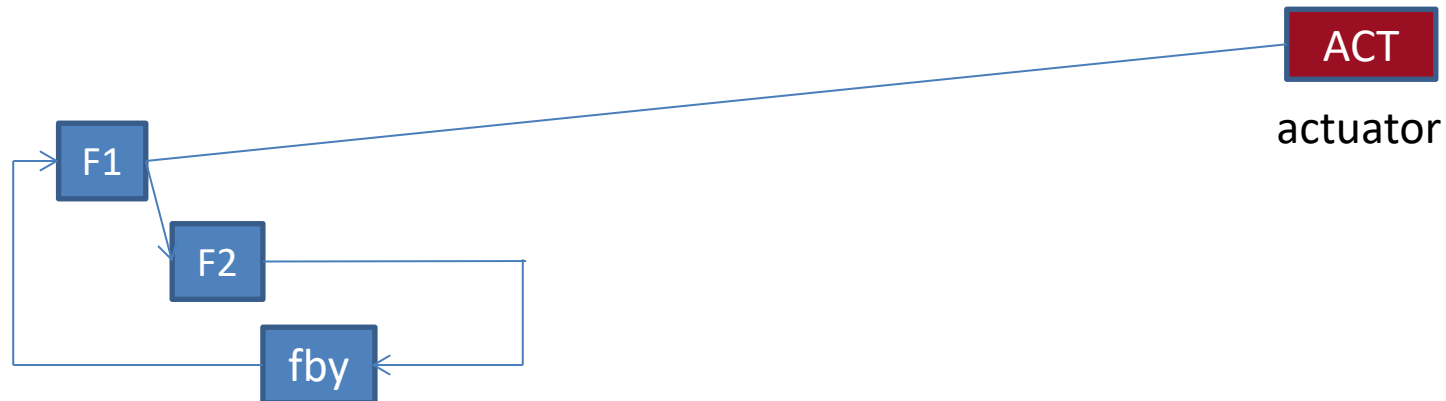
# Un exemple plus compliqué

- Modèle de contrôleur de moteur à essence
- Exécution cyclique
  - 1 cycle synchrone par rotation du moteur
  - Calcul de l'allumage du cycle suivant



# Un exemple plus compliqué

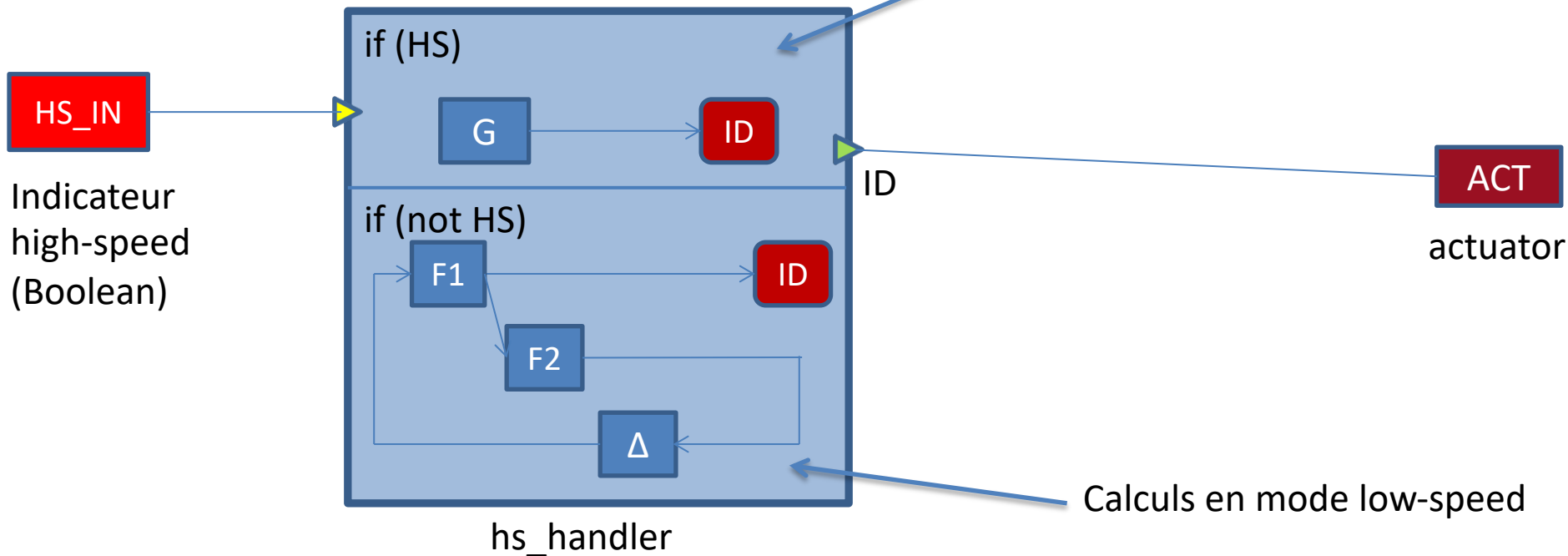
- F1 fait à la fois de la capture de données et du calcul
- ACT est la mise à jour de l'angle d'allumage
  - actuateur





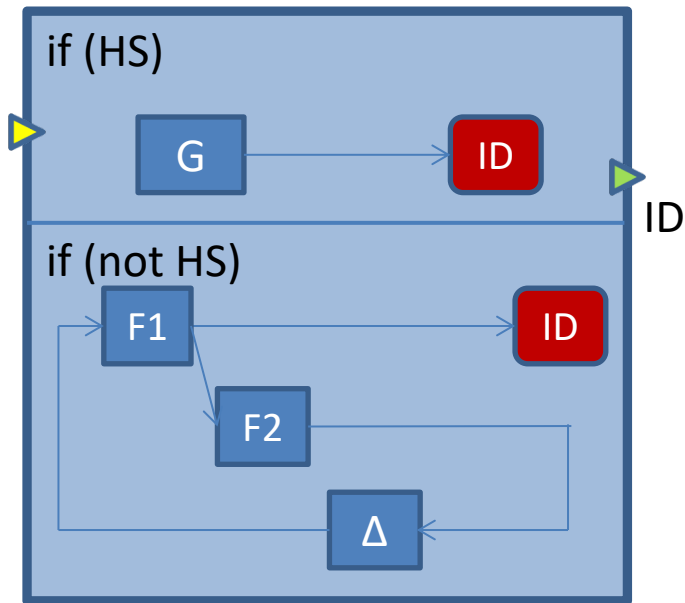
# Un exemple plus compliqué

- Vitesse variable du moteur => paliers de vitesse
  - HS -> moins de temps



# Un exemple plus compliqué

- Modèle Heptagon de `hs_handler` :



`hs_handler`

```
node hs_handler(hs:bool)
  returns (id:int)
let
  if hs then
    id = g() ;
  else
    var x,y : int; in
      y = 15 fby x ;
      id = f1(y) ;
      x = f2(id) ;
  end ;
tel
```

# Un exemple plus compliqué

- Modèle Heptagon du système

```
(* definition of hs_handler *)
```

```
...
```

```
node main () returns ()
```

```
var
```

```
  hs: bool ;
```

```
  id : int ;
```

```
let
```

```
  hs = read_bool(addr_hs) ;
```

```
  id = hs_handler(hs) ;
```

```
  () = act(id) ;
```

```
tel
```

# Un exemple plus compliqué

- Modèle Heptagon du système

```
(* definition of hs_handler *)
```

```
...
```

```
node main () returns ()
```

```
var
```

```
  hs: bool ;
```

```
  id : int ;
```

```
let
```

```
  hs = read_bool(addr_hs) ;
```

```
  id = hs_handler(hs) ;
```

```
  () = act(id) ;
```

```
tel
```

## Convention:

Une spécification "système" n'a pas des entrées et des sorties. Elles sont acquises par des fonctions de lecture de capteurs et d'écriture d'actionneurs

# Un exemple plus compliqué

- Constante `addr_hs` :

```
const addr_hs:int = 0x2000 (* global constant *)
```

```
(* definition of hs_handler *)
```

```
...
```

```
node main () returns ()
```

```
var
```

```
  hs: bool ;
```

```
  id : int ;
```

```
let
```

```
  hs = read_bool(addr_hs) ;
```

```
  id = hs_handler(hs) ;
```

```
  () = act(id) ;
```

```
tel
```

# Un exemple plus compliqué

- Définitions externes :

```
open Extern (* declaration of external functions *)
```

```
const addr_hs:int = 0x2000 (* global constant *)
```

```
(* definition of hs_handler *)
```

```
...
```

```
node main () returns ()
```

```
var
```

```
  hs: bool ;
```

```
  id : int ;
```

```
let
```

```
  hs = read_bool(addr_hs) ;
```

```
  id = hs_handler(hs) ;
```

```
  () = act(id) ;
```

```
tel
```

# Un exemple plus compliqué

- Définitions externes :

`open Extern (* declaration of external functions *)`

- L'implémentation peut être fournie :

- En Heptagon : dans un fichier `extern.ept`
  - Fichier `abc.ept` -> "open Abc"
  - Impossible ici, car Heptagon n'a pas des fonctions d'I/O
- En C
  - Interface Heptagon dans un fichier `extern.epi`
    - » Fichier `abc.epi` -> "open Abc"
  - Source C :
    - » Deux fichiers de déclarations : `extern.h`, `extern_types.h`
    - » Des fichiers source avec l'implémentation, e.g. `extern.c`

# Un exemple plus compliqué

- Implémentation en C. Fichier extern.epi :

```
fun read_bool(addr:int) returns (value:bool)
fun f1 (i:int) returns (o:int)
fun f2 (i:int) returns (o:int)
fun g () returns (o:int)
fun act (i:int) returns ()
```

- Toutes les fonctions utilisées, mais non-définies en Heptagon, doivent être déclarées dans un fichier .epi inclus
- On peut déclarer fonctions, nodes, types, const.
- Le(s) fichiers .epi doivent être compilés :

```
heptc -c extern.epi
```

- Cela produit le fichier extern.epci



# Un exemple plus compliqué

- Implémentation en C. Fichier extern.h:

```
typedef struct { int value ; } Extern__read_bool_out ;
typedef struct { int o ; } Extern__f1_out ;
typedef struct { int o ; } Extern__f2_out ;
typedef struct { int o ; } Extern__g_out ;
typedef struct { } Extern__act_out ;

void Extern__f1_step(int i, Extern__f1_out*_out) ;
void Extern__f2_step(int i, Extern__f2_out*_out) ;
void Extern__g_step(Extern__g_out*_out) ;
void Extern__read_bool_step(int addr,
                             Extern__read_bool_out*_out) ;
void Extern__act_step(int addr, Extern__act_out*_out) ;
```

# Un exemple plus compliqué

- Implémentation en C. Fichier extern.h:

```
typedef struct { int value ; } Extern__read_bool_out ;  
typedef struct { } Extern__act_out ;
```

```
void Externc__f1_step(int i,Externc__f1_out*_out) ;
```

- Les noms C suivent les mêmes conventions que le code généré par heptc
- On peut définir : fonctions, noeuds, types, constantes

# Un exemple plus compliqué

- Implémentation en C. Fichier extern.c:

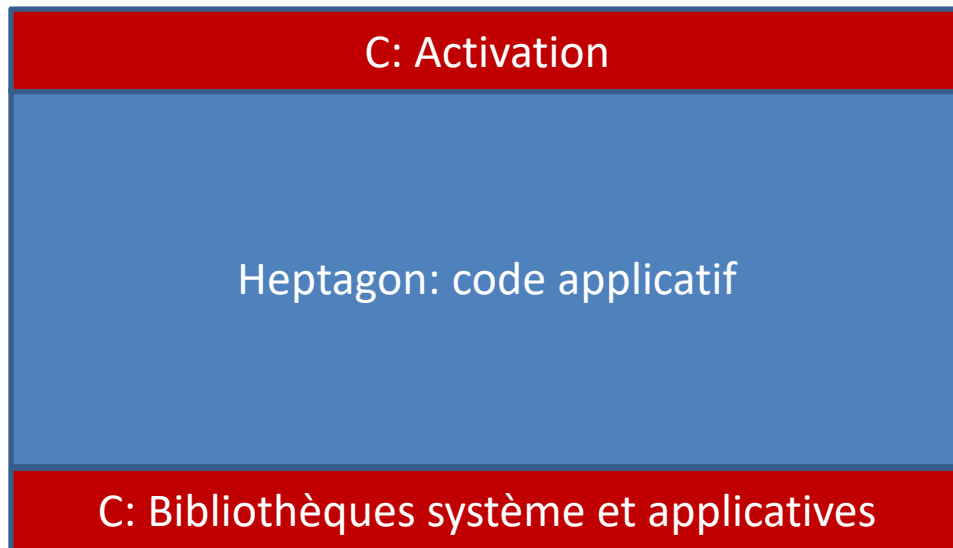
```
void Extern__read_bool_step(int addr,
                             Extern__read_bool_out*_out) {
    printf("read_bool(%d):",addr) ; fflush(stdout) ;
    scanf("%d",&(_out->value)) ;
}
void Extern__f2_step(int i, Extern__f2_out*_out) {
    _out->o = i + 5 ;
    printf("F2(%d)=%d\n",i,_out->o) ;
}
void Extern__g_step(Extern__g_out*_out) {
    static int s = 0 ;
    s += 7 ;
    _out->o = s ;
    printf("G()=%d\n",_out->o) ;
}
...
```

# Un exemple plus compliqué

- Compilation :
  - D'abord les interfaces Heptagon .epi
  - Ensuite les fichiers .ept
  - Ensuite, compilation du code C :
    - code C de externc.c (et autres bibliothèques)
    - code C généré pour les fichiers .ept
    - le code C avec la fonction main (appel cyclique, comme la dernière fois)

# Interface Heptagon-C

- Structure d'une application Lustre/Heptagon



main.c

Fichiers .c et .h générés  
par heptc

- Implantations de fichiers .epi
- Autres bibliothèques appelées depuis main.c ou depuis l'implantation de fichiers .epi

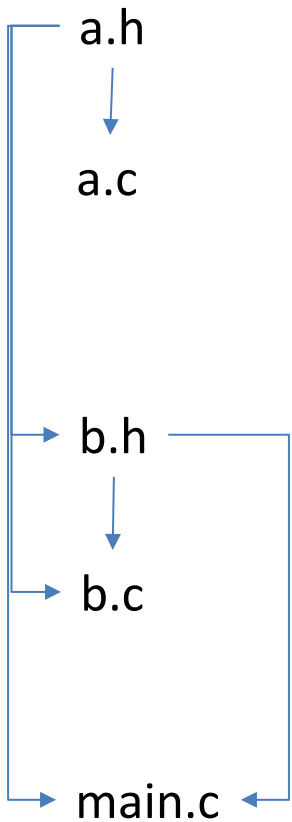
# Préparation du TP

- Programmer et exécuter trois objectifs :
  - Obj1: Exemple du transparent 29 (rcounter)
    - À chaque cycle, lecture de rst depuis le clavier (demander un booléen 0/1)
  - Obj2: L'exemple du transparent 47/54
    - Lecture de HS depuis le clavier, à chaque cycle (booléen 0/1)
    - Compilation et execution cylique
      - Implémentation des fonctions de externc.epi en suivant l'exemple du transparent 59
        - »  $f1(x) = x+5$
        - »  $f2(x) = x + 100$
        - »  $g() =$  counter starting at 300 and which advances by steps of 50
  - Obj3: Réécrire l'exemple du transparent 47/54 en remplaçant dans hs\_handler "if" par "when" et "merge"
    - Expression native flot de données

# Compilation

- Compilation séparée en C

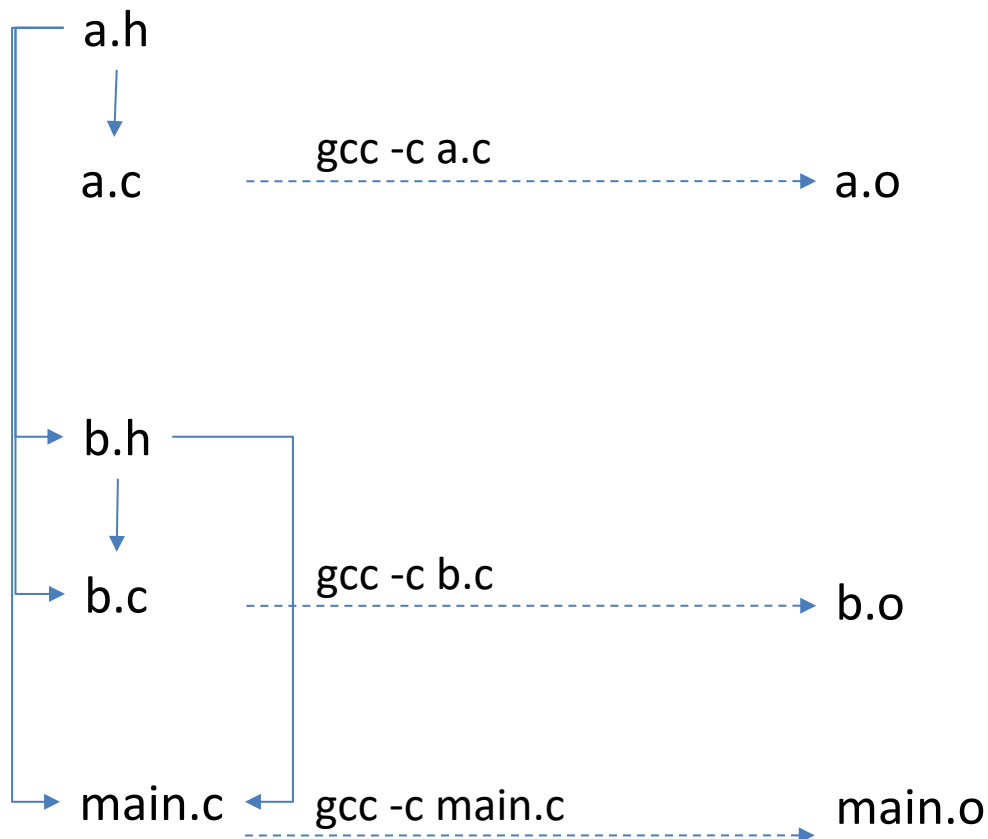
main.exe



# Compilation

- Compilation séparée en C

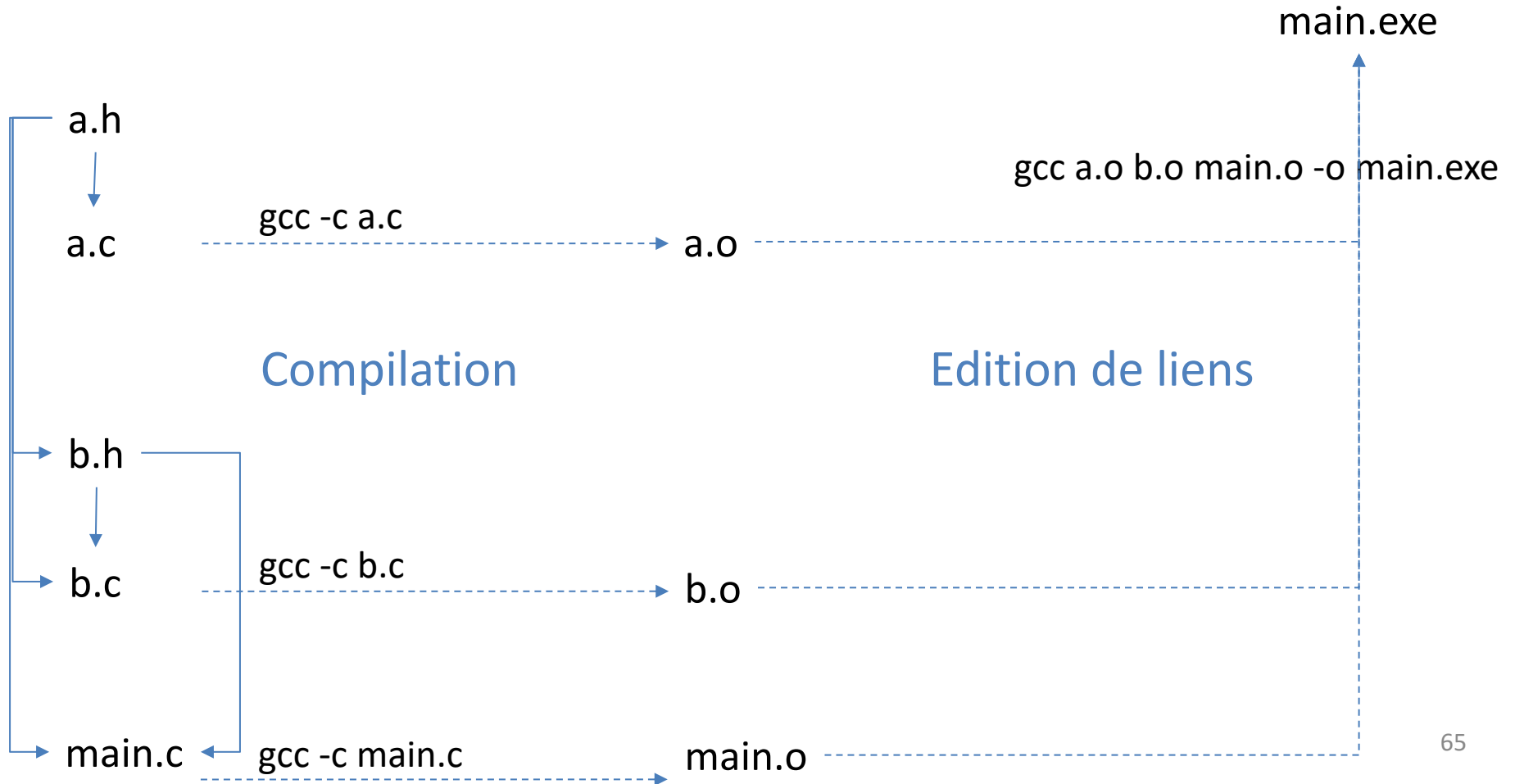
main.exe





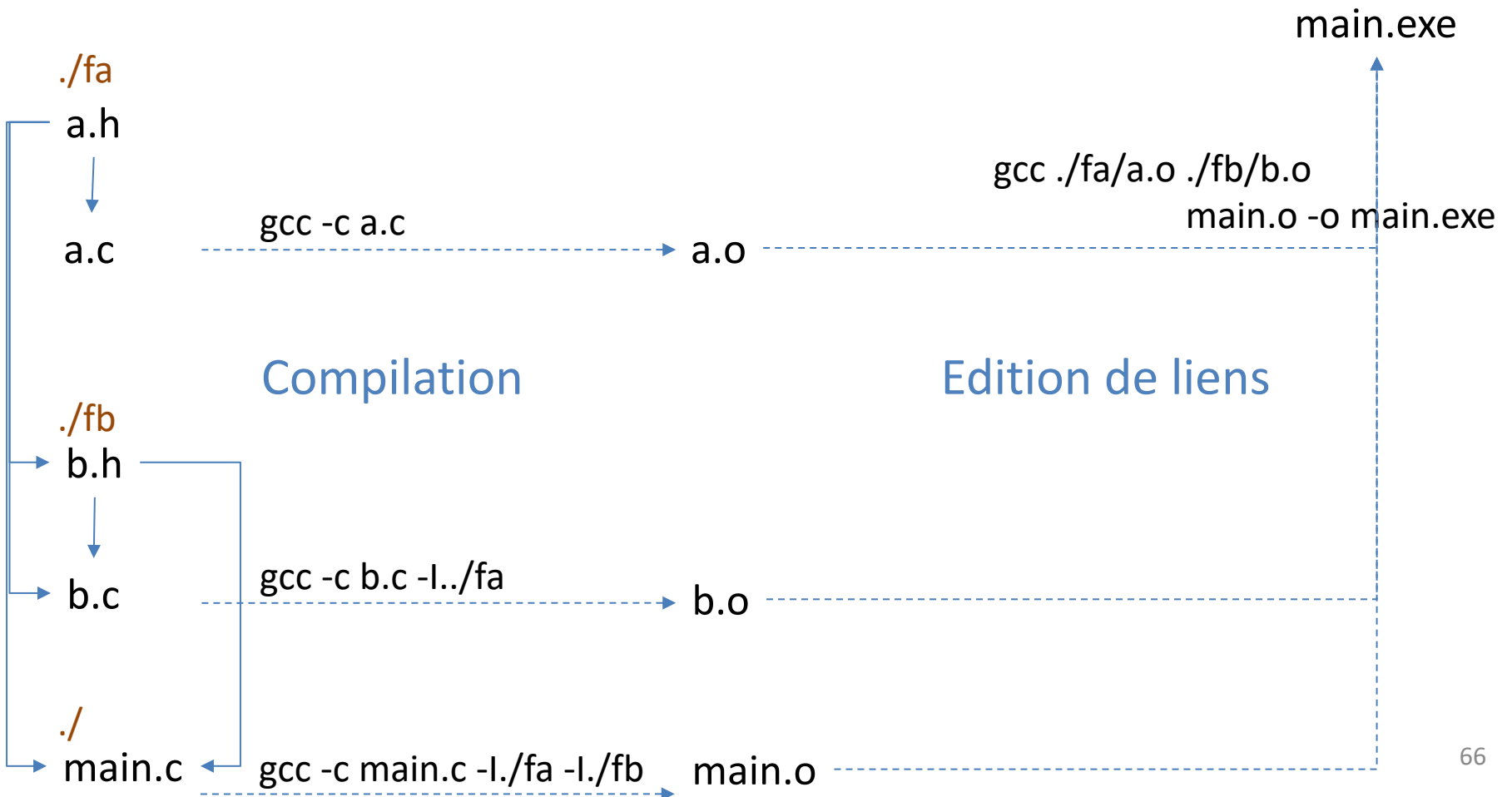
# Compilation

- Compilation séparée en C



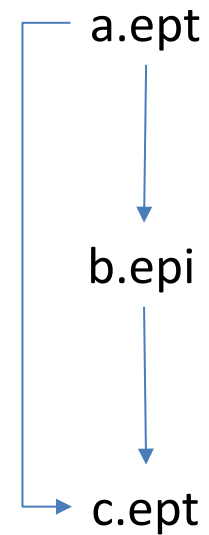
# Compilation

- Compilation séparée en C



# Compilation

- Compilation Heptagon+C

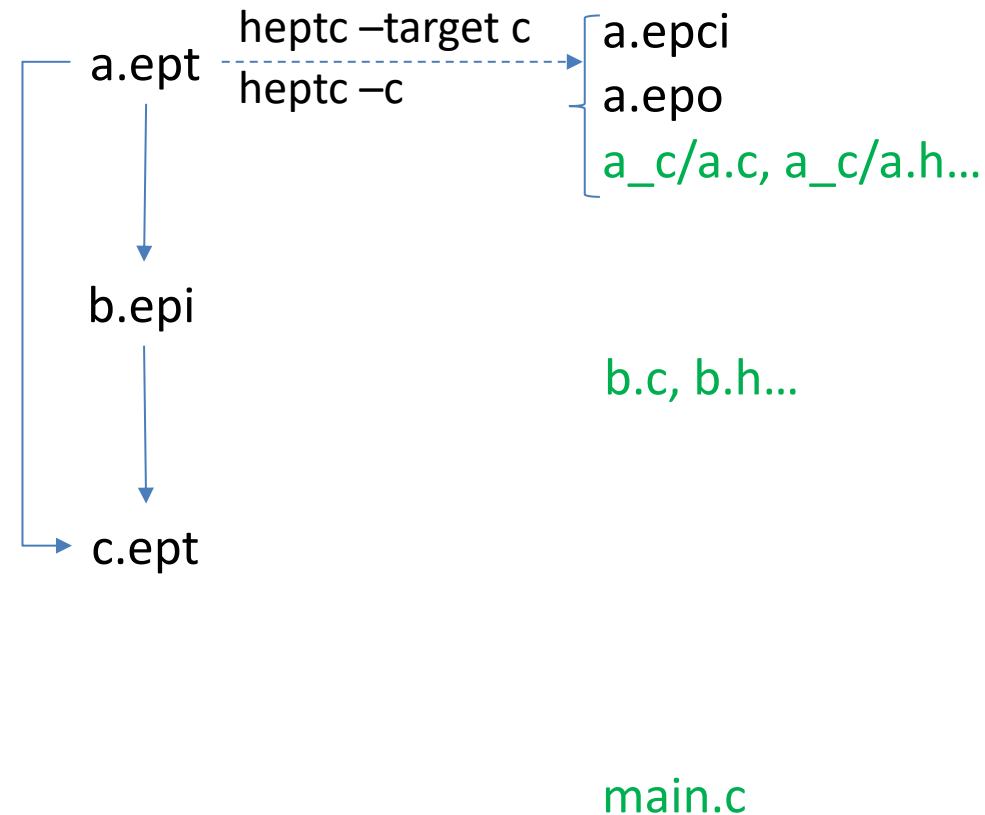


b.c, b.h...

main.c

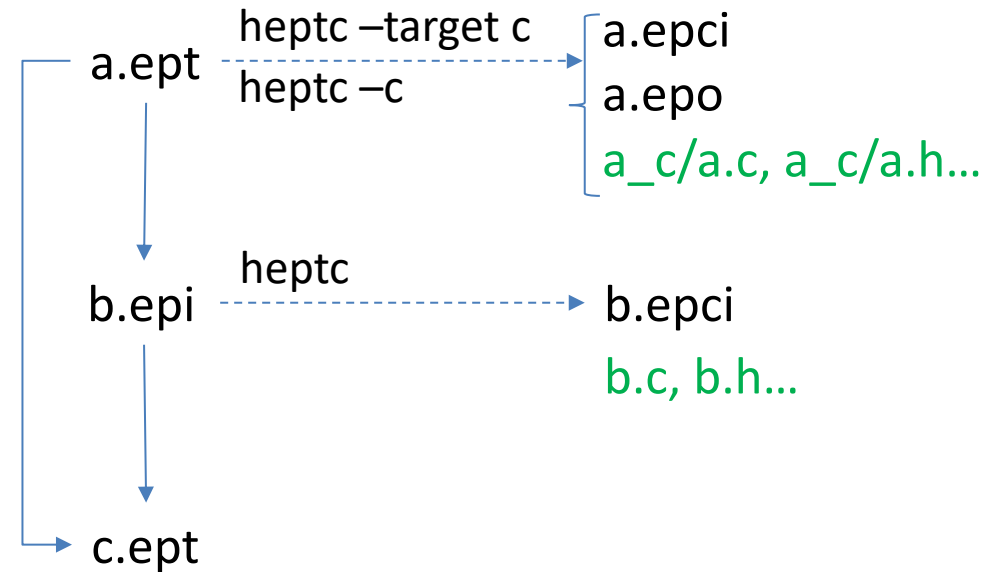
# Compilation

- Compilation Heptagon+C



# Compilation

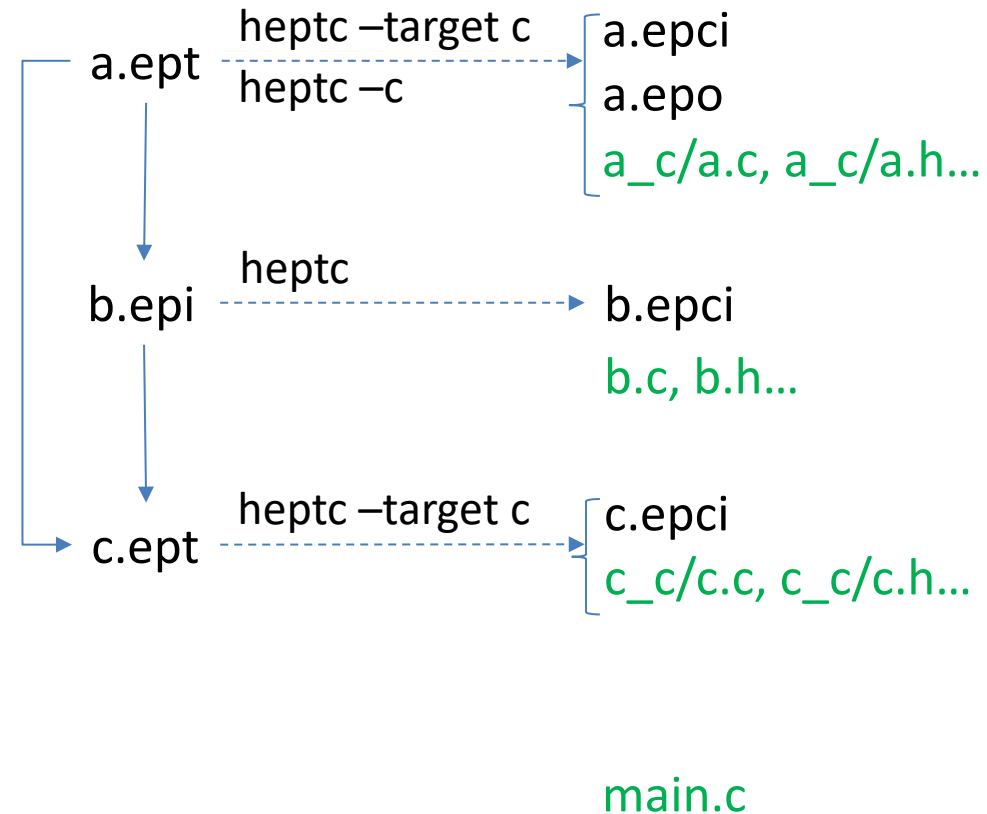
- Compilation Heptagon+C



main.c

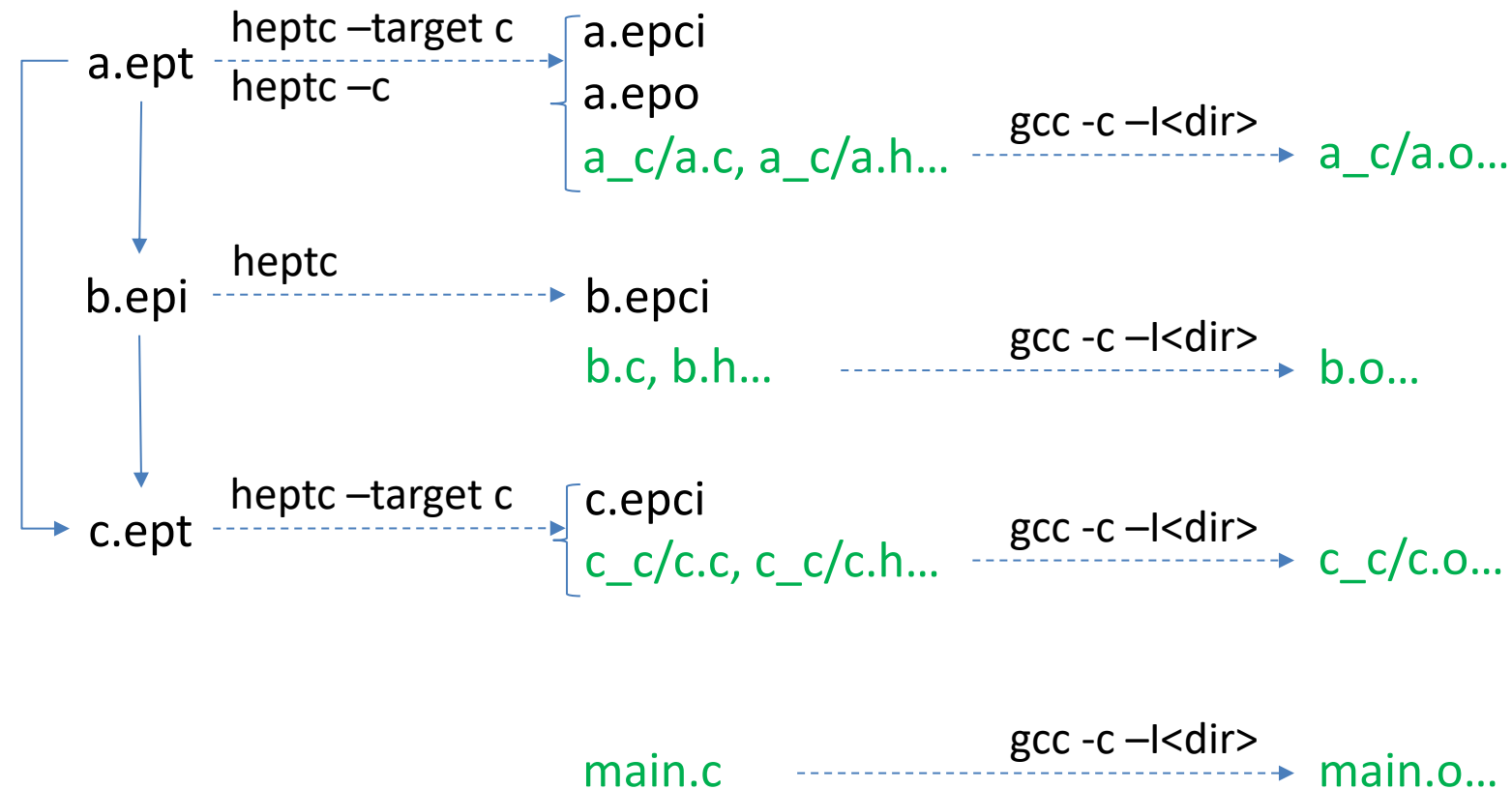
# Compilation

- Compilation Heptagon+C



# Compilation C

- Compilation Heptagon+C



# Compilation C

- Compilation Heptagon+C

