Une approche synchrone à la conception de systèmes embarqués temps réel

Dumitru Potop-Butucaru

dumitru.potop@inria.fr

cours EIDD, 2025, 8ème séance

Evaluation (rappel)

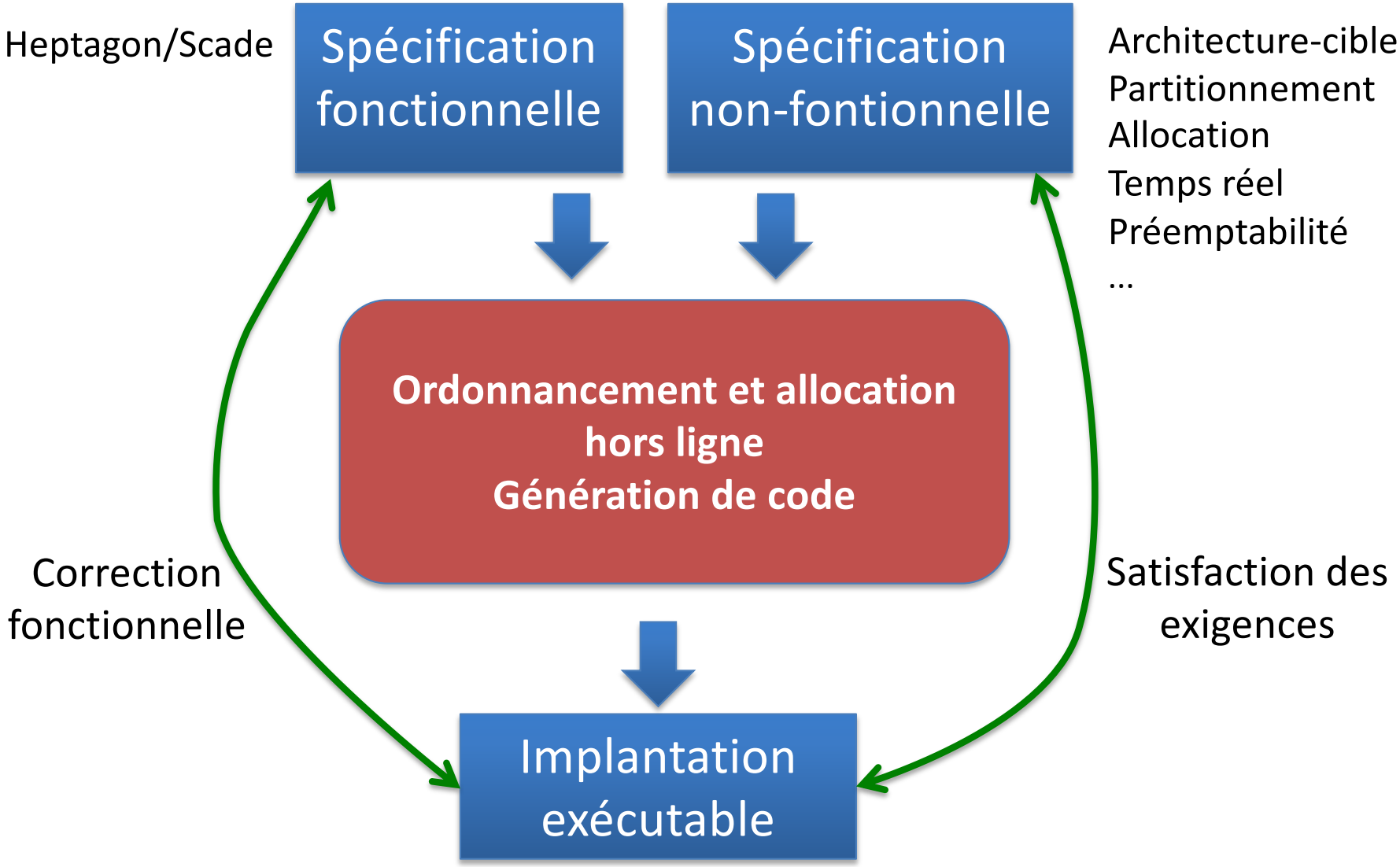
- Contrôle continu, sur le TP
  - Évaluation par problème résolu
  - Dernière séance dédiée aux rendus, et au retour des kits de travail Raspberry Pi
    - Sans retour : -40% sur la note

- Rattrapage – le tout (continu+examen) condensé sur 1-2 jours (je ne recommande pas)

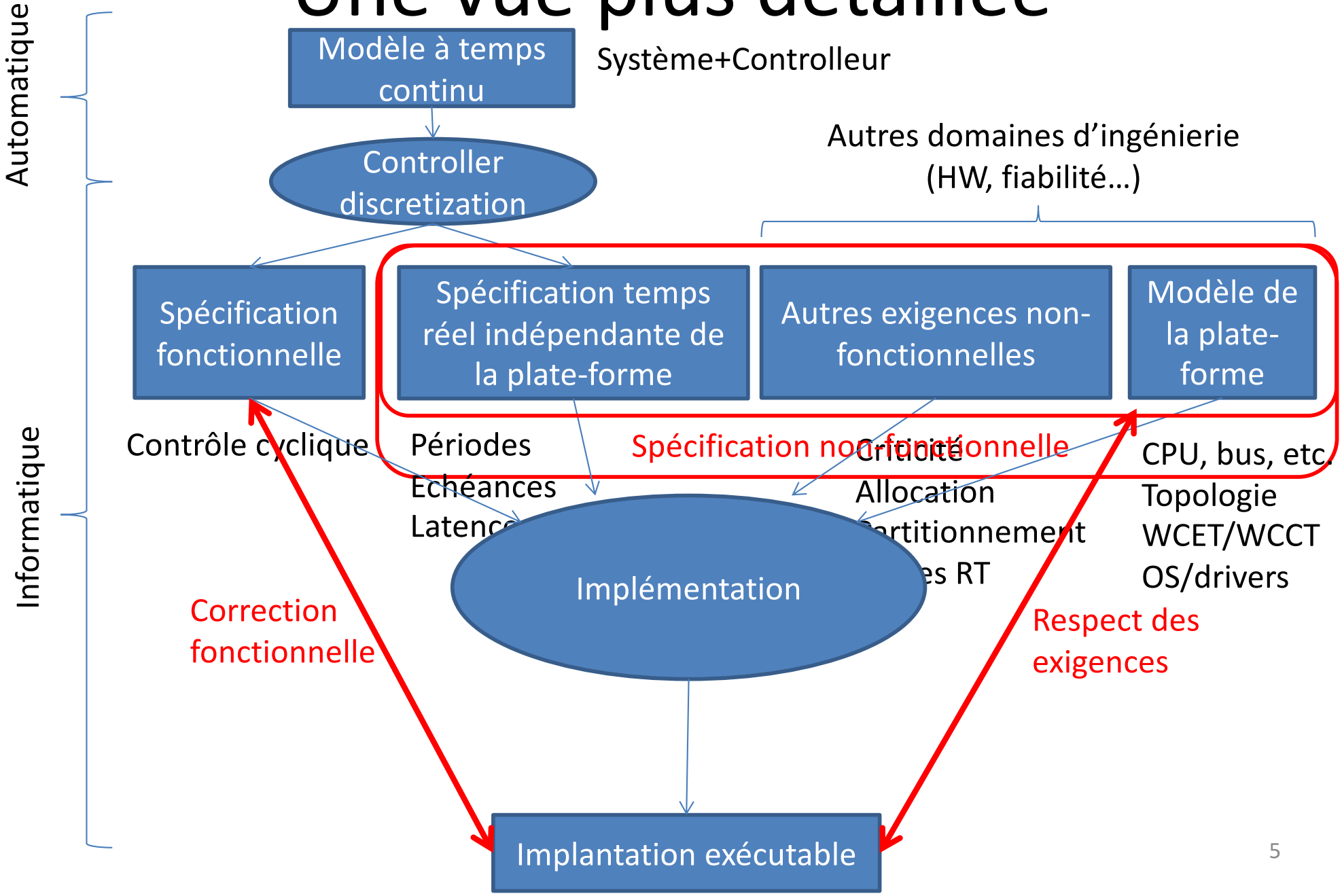
Contenu du cours

- Cours précédents
  - Spécification fonctionnelle de haut niveau
  - Plate-forme d'exécution multi-cœurs
  - Théorie de l'ordonnancement temps réel
  - Implantation manuelle (applicatif et ordonnanceur) sur multi-cœur
- Aujourd'hui (cours) : Automatisation
  - Quand automatiser ?
  - L'outil Lopht

Systèmes embarqués: implémentation



Une vue plus détaillée



Implantation de systèmes embarqués

- Des étapes importantes sont encore artisanales
  - Manuelles ou non-formalisées
    - Certaines qui pourraient être automatisées avec des outils de niveau industriel (modulo arguments de certification)
      - Discrétisation de modèles en temps continu
      - Construction des taches à partir de bouts de code C
      - Allocation et ordonnancement des tâches pour assurer la correction fonctionnelle
      - Programmer des comportements temporels complexes
    - Certaines qui sont encore non-automatisables
      - Définition de modèles de la plate-forme d'execution permettant une analyse temporelle précise et sûre
      - Allocation & ordonnancement garantissant le respect des exigences temps reel sans hypothèses non-vérifiables
        - » **On doit toujours analyser le système après construction**
  - Sujettes à erreurs, consummation de temps et de ressources
  - Finalement, il faut massivement tester pour déterminer si le systeme fonctionne correctement

L'artisanat ne passe plus à l'échelle

- La complexité explose
  - Beaucoup de ressources (e.g. multi-/many-cores)
  - Beaucoup de logiciels (plus de tâches, plus de logiciel système)
  - HW, SW plus complexe
    - Réseaux de communication (NoCs, hiérarchies mémoires, etc.), Ordonnanceurs hiérarchiques (virtualisation), Tâches dépendantes…
- Embarqué hautes performances: l'efficacité devient critique
  - Filtres parallélisés (ML, analyse vibratoire) = tâches dépendantes
    - Mauvais ordonnancement => perte rapide de performance
  - Tous les paramètres influent sur la performance
    - Allocation mémoire
    - Allocation et ordonnancement des tâches
    - Synchronisations, communications…
- Déléguer ces tâches à l'OS => imprédictibilité temporelle
  - Cohérence de caches, GPUs, allocation&ordo dynamiques…

Il faut automatiser

Il faut automatiser

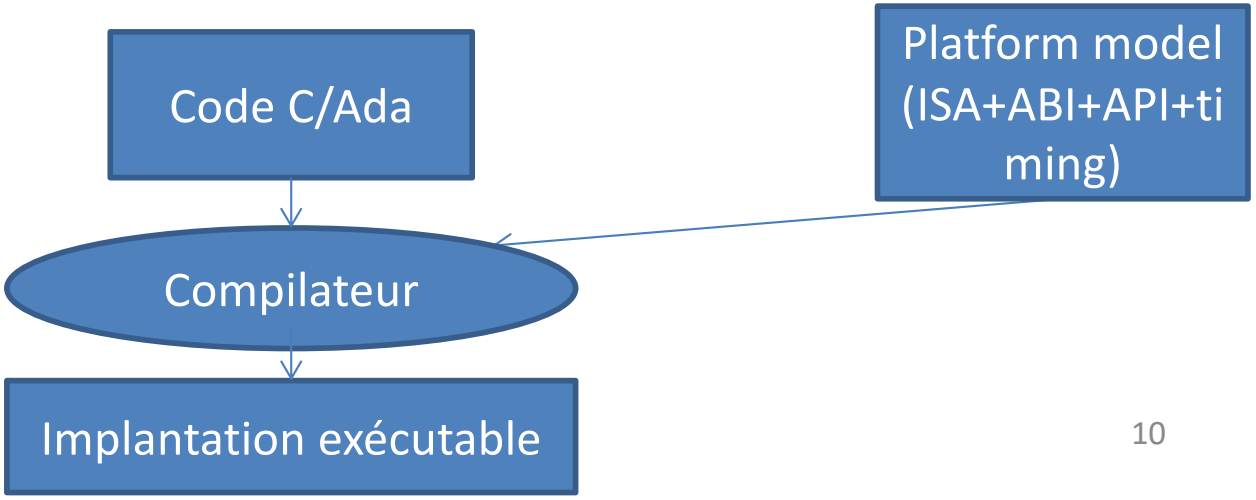
- Cela demande la formalisation complète
  - De la spécification fonctionnelle
  - Des exigences non-fonctionnelles
  - Du modèle de la plate-forme
    - Le modèle est-il une abstraction sûre de la plate-forme ?
  - De la correction de l'implantation
- Besoin d'algorithmes efficaces d'analyse et de synthèse

La standardisation est un prérequis

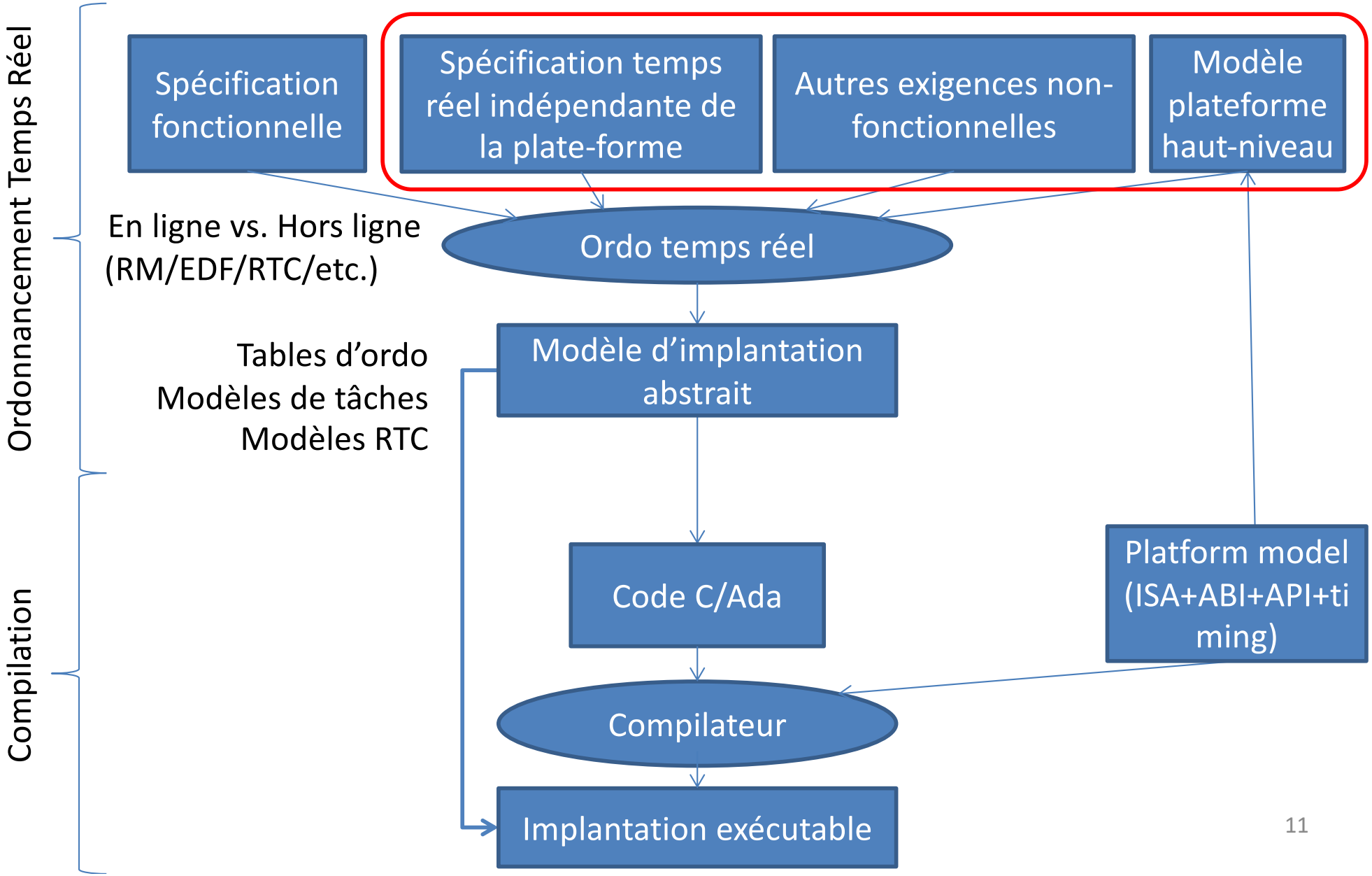
(mutualiser les ressources => économiquement viable)

Le grand succès de l'automatisation: la compilation

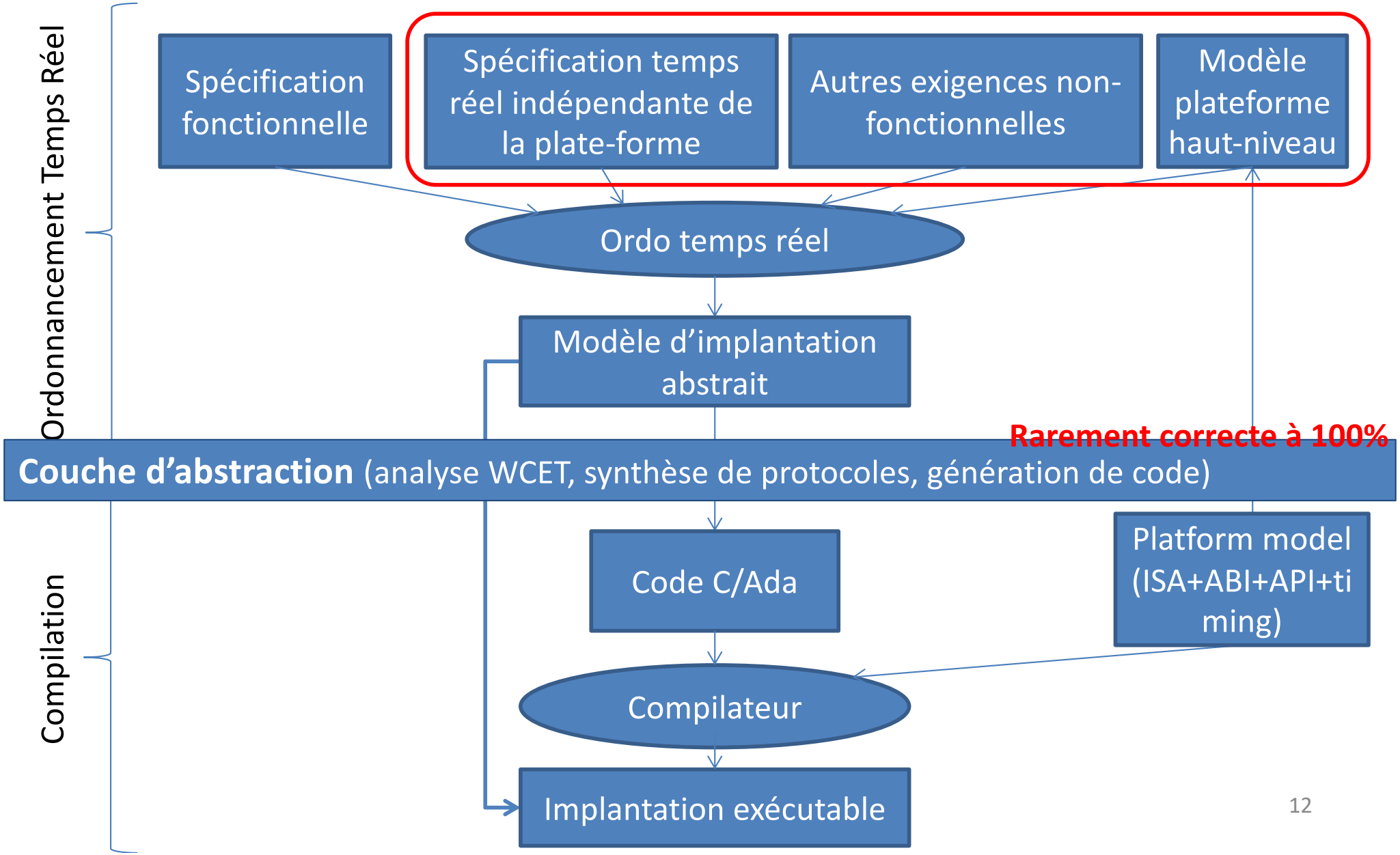
- Niveaux bas du flot d'implantation embarqué
- Rendu possible par la standardisation précoce des langages de programmation et des plates-formes d'exécution (ISAs, ABIs, APIs).
- Il n'y a presque plus de codage assembleur, même dans l'embarqué



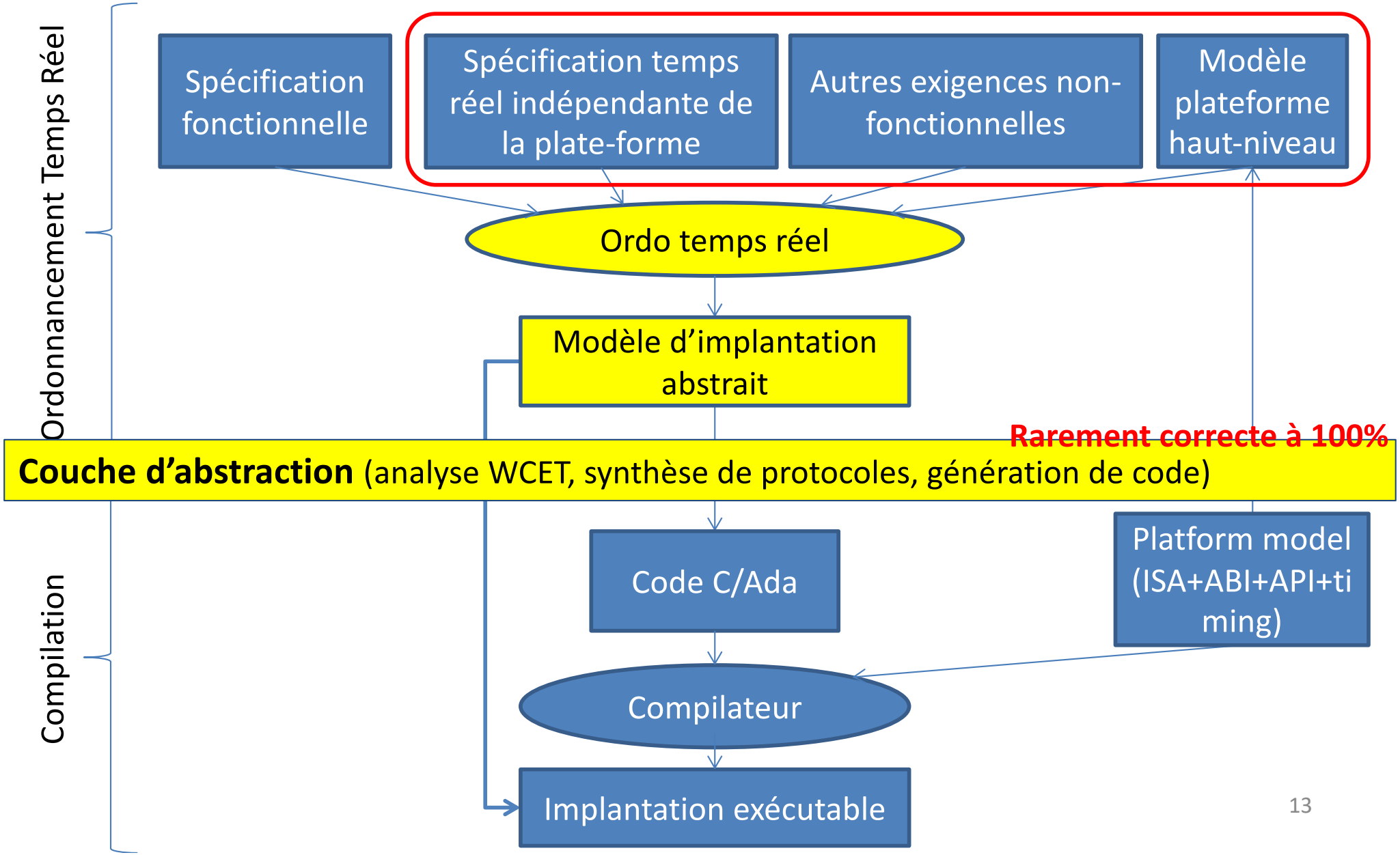
Ordo temps réel vs. Compilation



Ordo temps réel vs. Compilation



Difficulté : automatisation complète



Difficulté : automatisation complète

- Historiquement difficile (absence de standardisation)
- Plus récemment: langages de spécification fonctionnelle et non-fonctionnelle
  - Spécification de systèmes dynamiques
    - Simulink, LabView, Scade, PsyC, etc.
    - Géneration de code assurant la correction fonctionnelle et **partiellement** le respect des exigences
  - Plates-formes d'exécution pour le temps réel
    - ARINC 653, AUTOSAR, TTP/TTEthernet, TSN etc.
    - Many-cores (certains), multi-cores (Aurix)…
  - Langages d'ingénierie système
    - SysML, AADL…
    - Temps réel, partitionnement, criticalité, allocation…

L'outil automatique idéal

- Passer de spécification à code compilable et exécutable
  - En garantissant la correction fonctionnelle et non-fonctionnelle
    - **Plus besoin de test/analyse intensive**
  - De manière complètement automatique
    - Sans intervention humaine durant la transformation
  - Entrées et sorties formalisées
    - Spécification fonctionnelle et non-fonctionnelle, modèle d'implantation
    - Garanties de correction formelles
    - Sans cassure dans la chaîne de formalisée
      - Preuve possible
  - Traçabilité des causes d'échec
    - Erreurs fonctionnelles et limites non-fonctionnelles
    - Méthode de conception essai-erreur
  - **Rapide et efficace**
    - **Modélisation précise de la plates-forme d'exécution et de l'application**
    - **Utilisation d'heuristiques rapides**
      - **Compilation, ordonnancement temps réel, langages synchrones…**
      - **Les méthodes « exactes » ne passent pas à l'échelle [** https://hal.inria.fr/hal-01250010/document**]**

Commençons par le cas simple

- Systèmes à allocation statiques des ressources
  - Analyse de temps d'exécution facilitée
    - Sans pénaliser la performance
    - Outils existants pour WCET/WCRT
    - Systèmes bien compris théoriquement
  - Classes d'applications importantes en pratique
    - Contrôle de systèmes embarqués critiques (e.g. transportation)
    - Traitement de signal/image
  - Modèle d'implantation formel : tables d'ordonnancement
    - Aussi appelées tables de reservations
    - Modèle partagé entre compilation et ordonnancement temps réel
      - » Réutilisation d'algorithmes facilitée

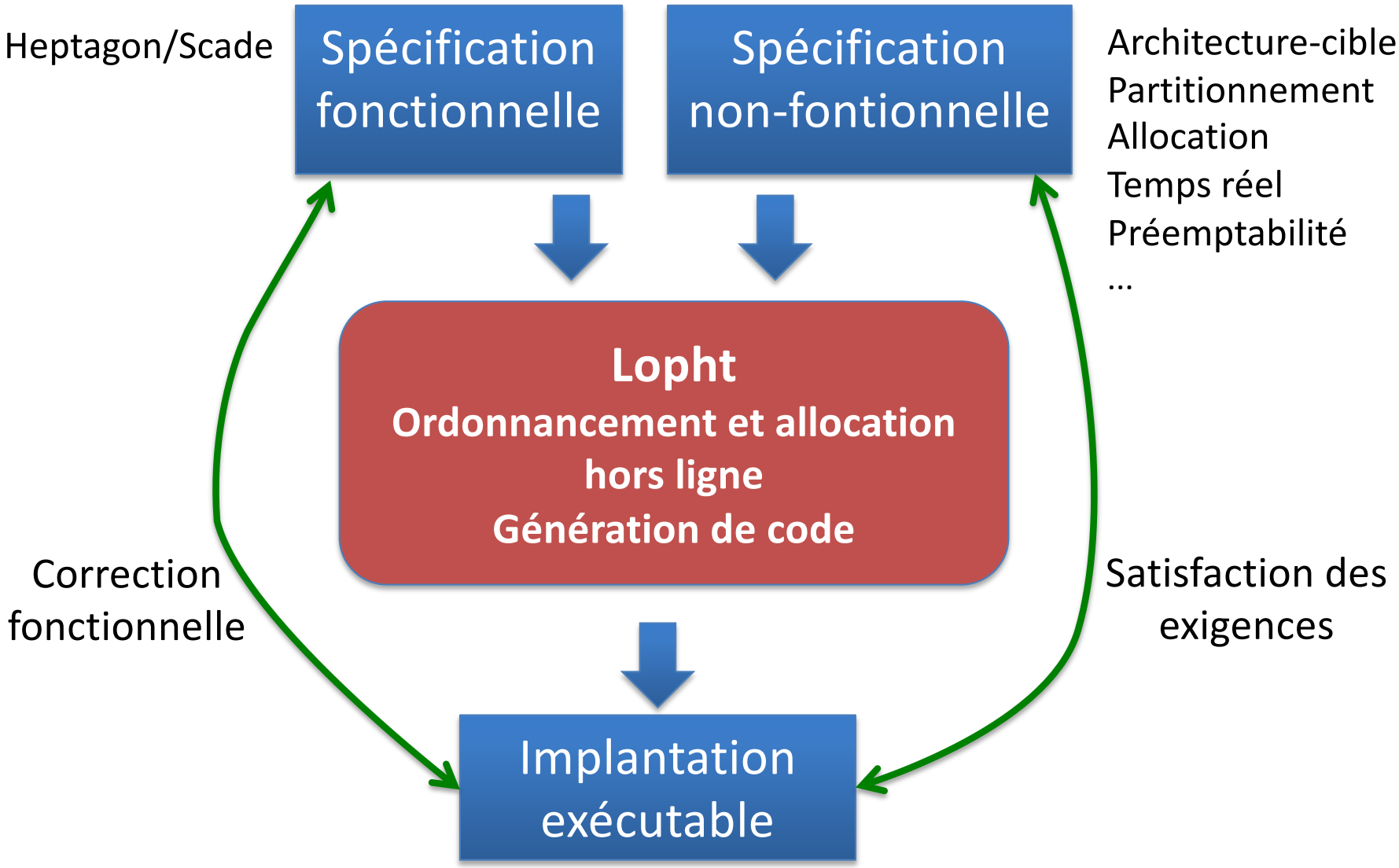
Travaux précédents

- Compilaton sur processeurs super-scalaires/VLIW/many-core
  - Objectif : optimisation, pas le respect d'exigences
  - Pas de garantie temps réel dur
  - Niveau de contrôle très (trop) fin
- Ordonnancement temps réel hors ligne
  - AAA/SynDEx (Sorel et al.)
    - Manquent : optimisations classiques (pipelining), tâches préemptables, allocation mémoire…
  - Simulink -> Scade -> TTA (Caspi et al.)
    - Manquent : Allocation, exécution conditionnelle, préemption, optimisations classiques (pipelining, memory bank allocation, etc.).
  - Prelude (Forget et al.)
    - Manquent : Partitionnement, échéances plus longues que les périodes, allocation mémoire…
  - PsyC/PharOS
    - Entrée de bas niveau (notre approche peut servir de front-end)

Le compilateur temps réel Lopht

- Compilateur pour systèmes temps réel statiquement ordonnées
  - Spécification fonctionnelle data-flow (e.g. Lustre/Scade/Heptagon)
  - Plates-formes d'exécution cibles
    - Systèmes répartis « time-triggered »
      - ARINC 653, TTEthernet
      - Génération complète de la configuration ARINC 653/TTEthernet, partition, synthèse de tout le code C
    - Multi-/many-cœurs (Kalray MPPA, Power T1042, ARM)
      - Génération du code C multi-thread, y compris synchronisations, allocation mémoire…
  - Applications testées
    - Contrôle de vol (Airbus, plusieurs versions)
    - Contrôle d'un moteur à réaction(Safran)
    - Modèle d'un lanceur spatial (Airbus DS)
    - Portes de métro (Alstom/IRT SystemX)

Automatisation



Lopht = mapping hors ligne

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

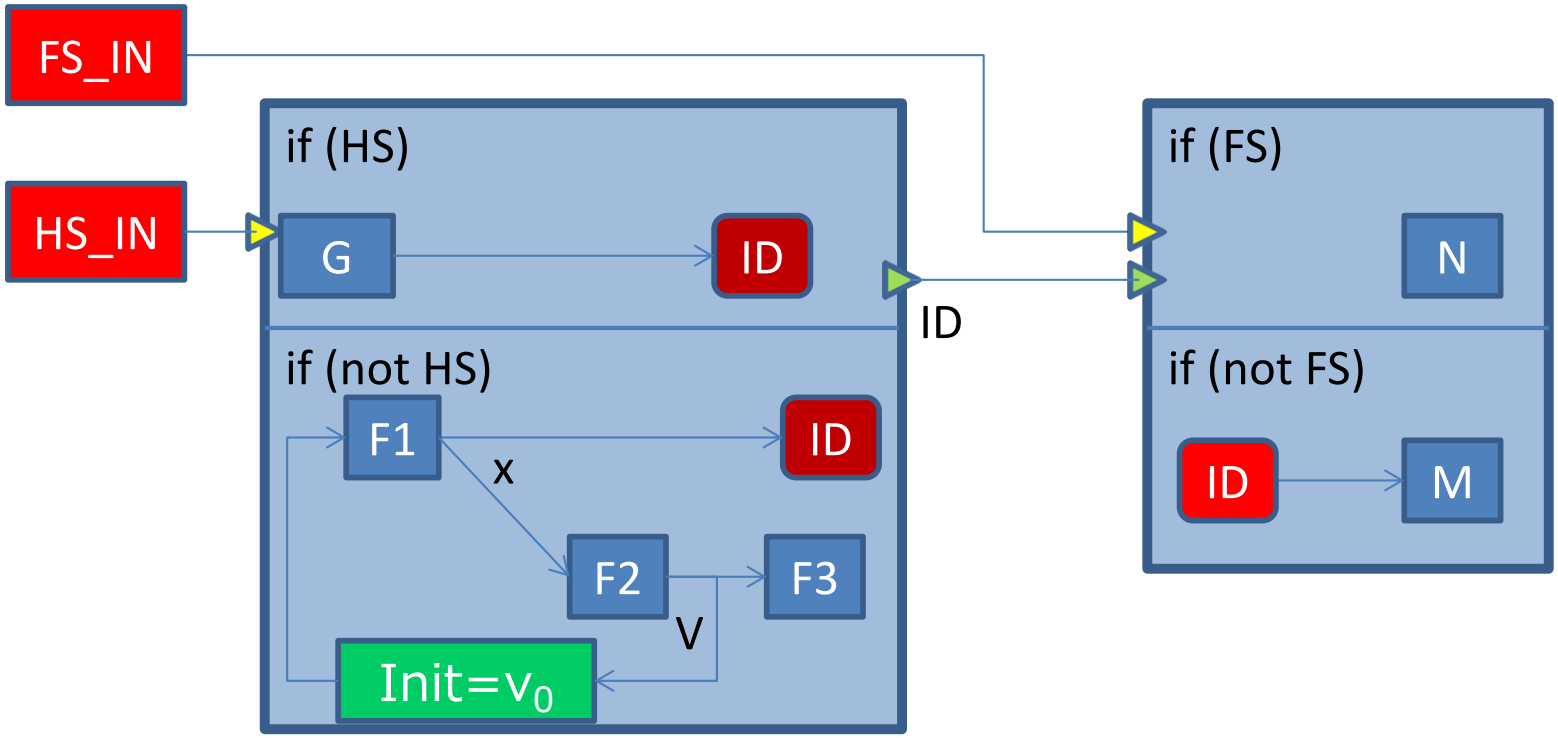
Heuristiques d'ordonnancement

- Décomposition du problème en plusieurs problèmes plus simples.
  - Allocation, puis ordonnancement
  - Allocation et ordonnancement d'un cycle de calcul, puis pipelinage
  - etc.
- Lopht
  - Ordonnancement d'un cycle de calcul
    - Puis pipelinage, si on veut

Exemple : spécification fonc.

- L'exemple de l'allumage moteur

HS: High speed mode
FS: Fail-safe mode



  - Spécification fonctionnelle : flot de données synchrone
    - Modèle d'exécution cyclique
    - 4 modes déterminés par deux interrupteurs indépendants

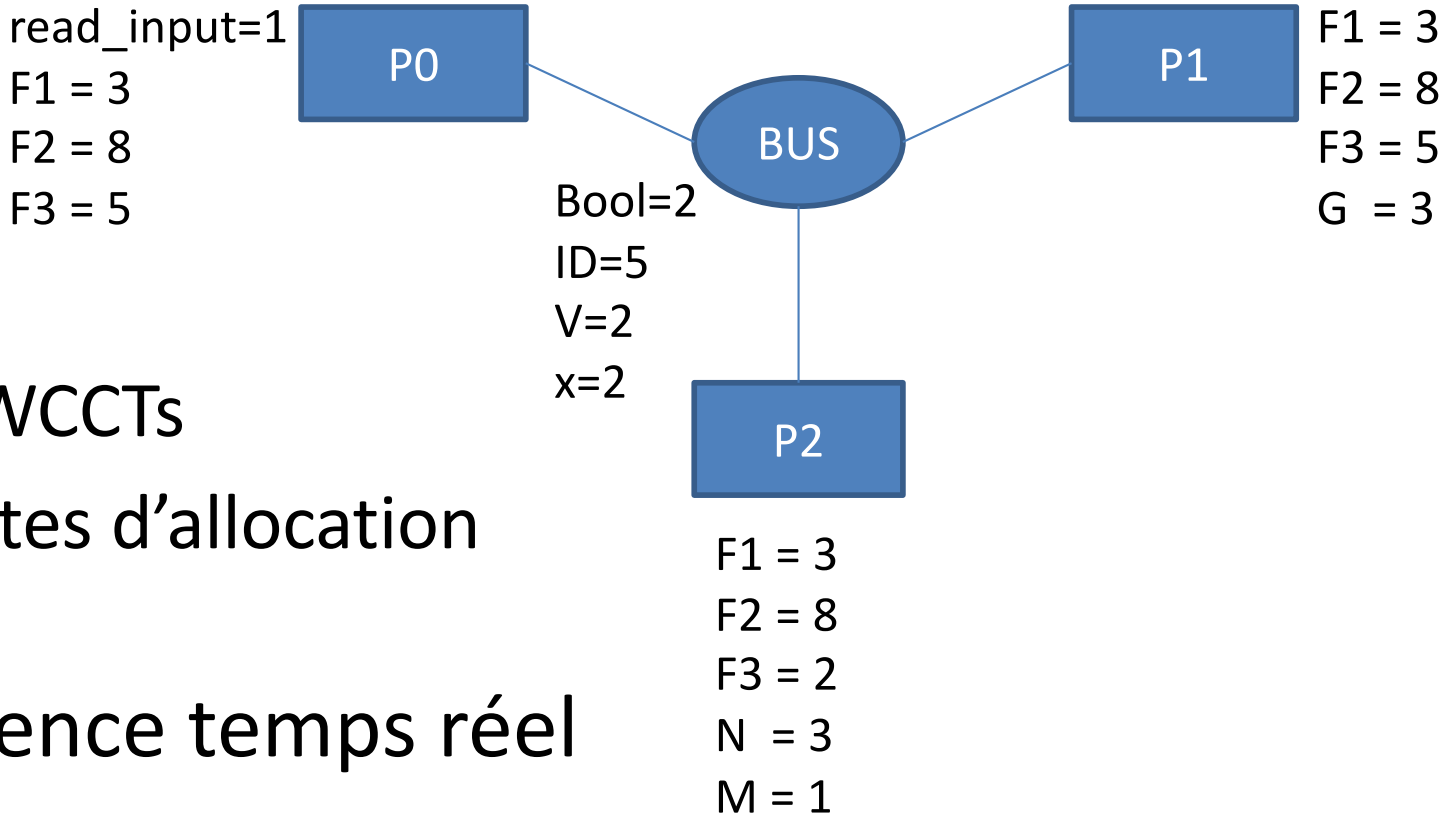
Exemple : spécification fonc.

- L'exemple de l'allumage moteur

```
node main() returns ()
var fs, hs: bool ; id, param: int ;
let
  fs = read_bool_sensor(fs_addr) ;
  hs = read_bool_sensor(hs_addr) ;
  if hs then id = g() ;
  else var x : int; in
    id = f1(0 fby x) ;
    x = f2(id) ;
  end ;
  if fs then
    param = default_param ;
  else
    param = id ;
  end ;
  () = act(param) ;
tel
```

Exemple : spécification non-fonc.

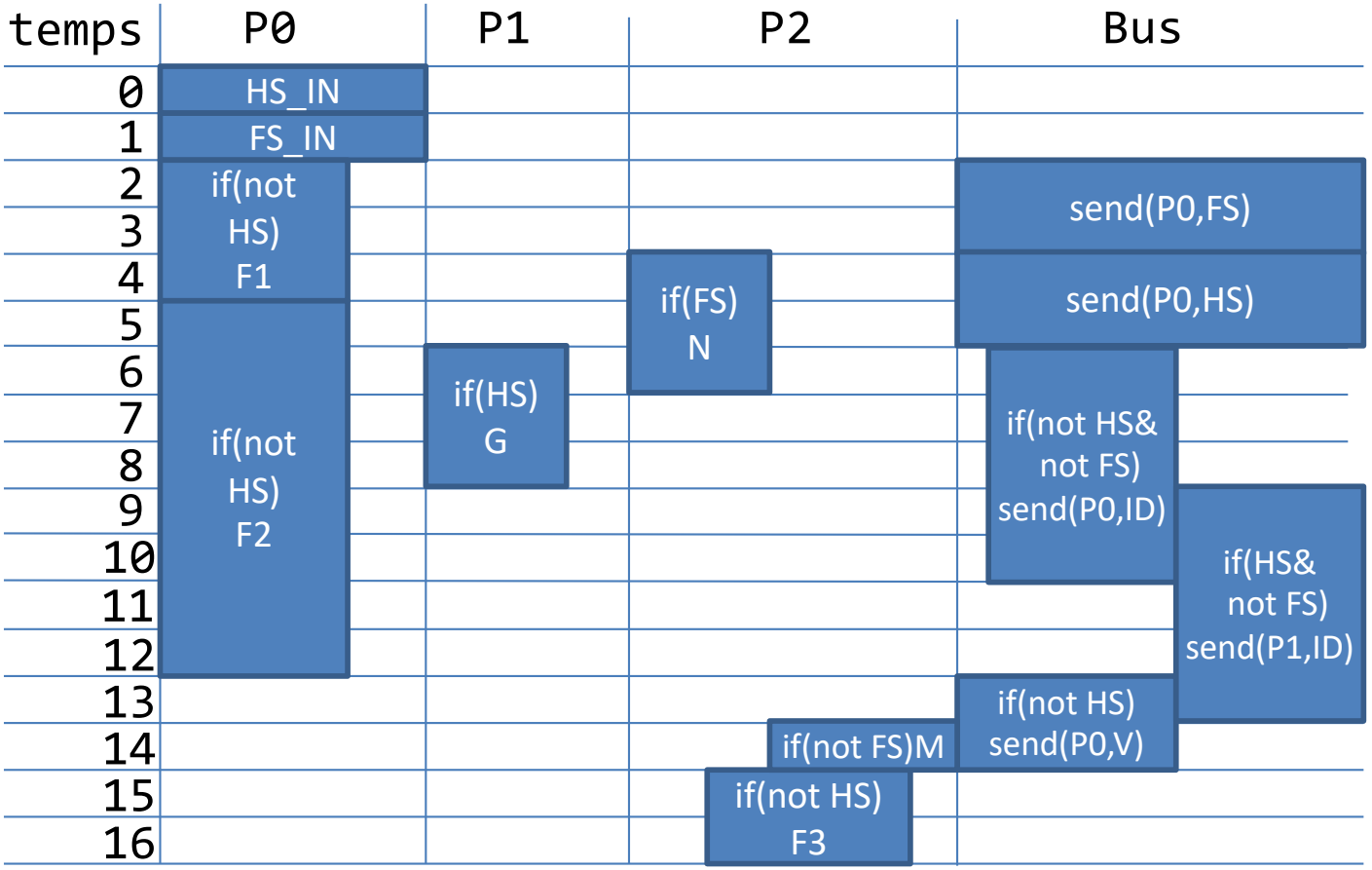
- Architecture, WCETs, WCCTs
  - 3 CPUs, 1 broadcast bus



  - WCETs/WCCTs
  - Contraintes d'allocation
- Pas d'exigence temps réel
- Objectif : optimisation de la durée d'un cycle

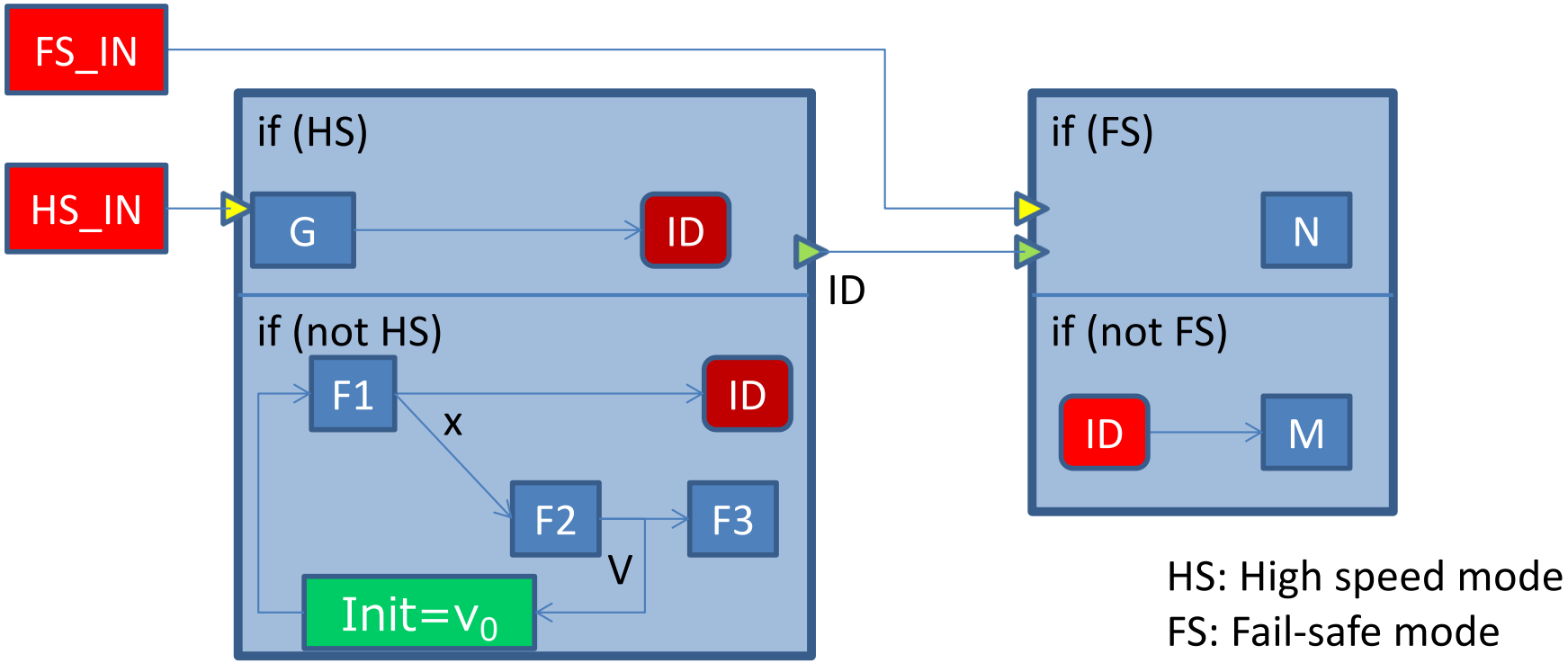
Exemple : ordonnancement résultant

- Table d'ordonnancement conditionnelle



Pré-traitement : mise à plat

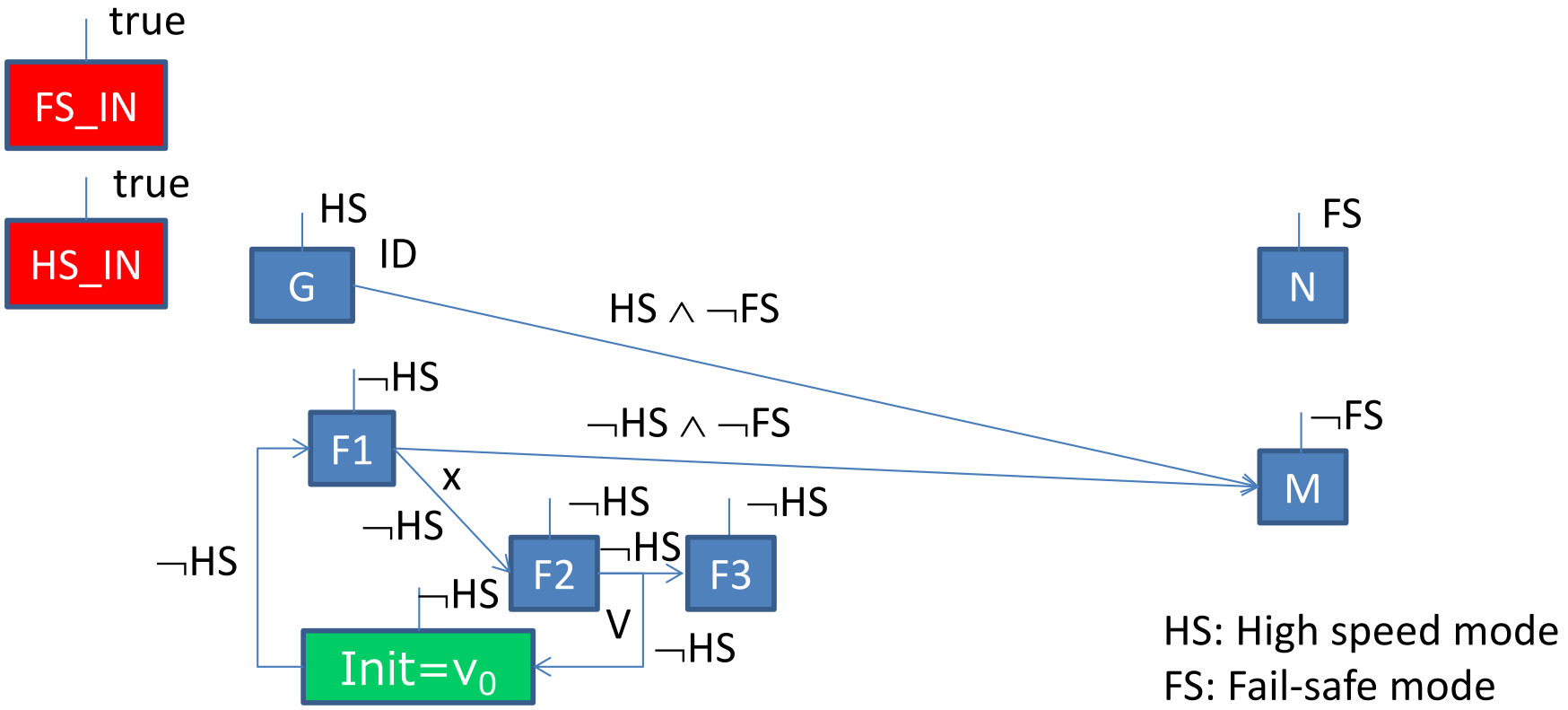
- Traduction en flot de données non hiérarchique



HS: High speed mode
FS: Fail-safe mode

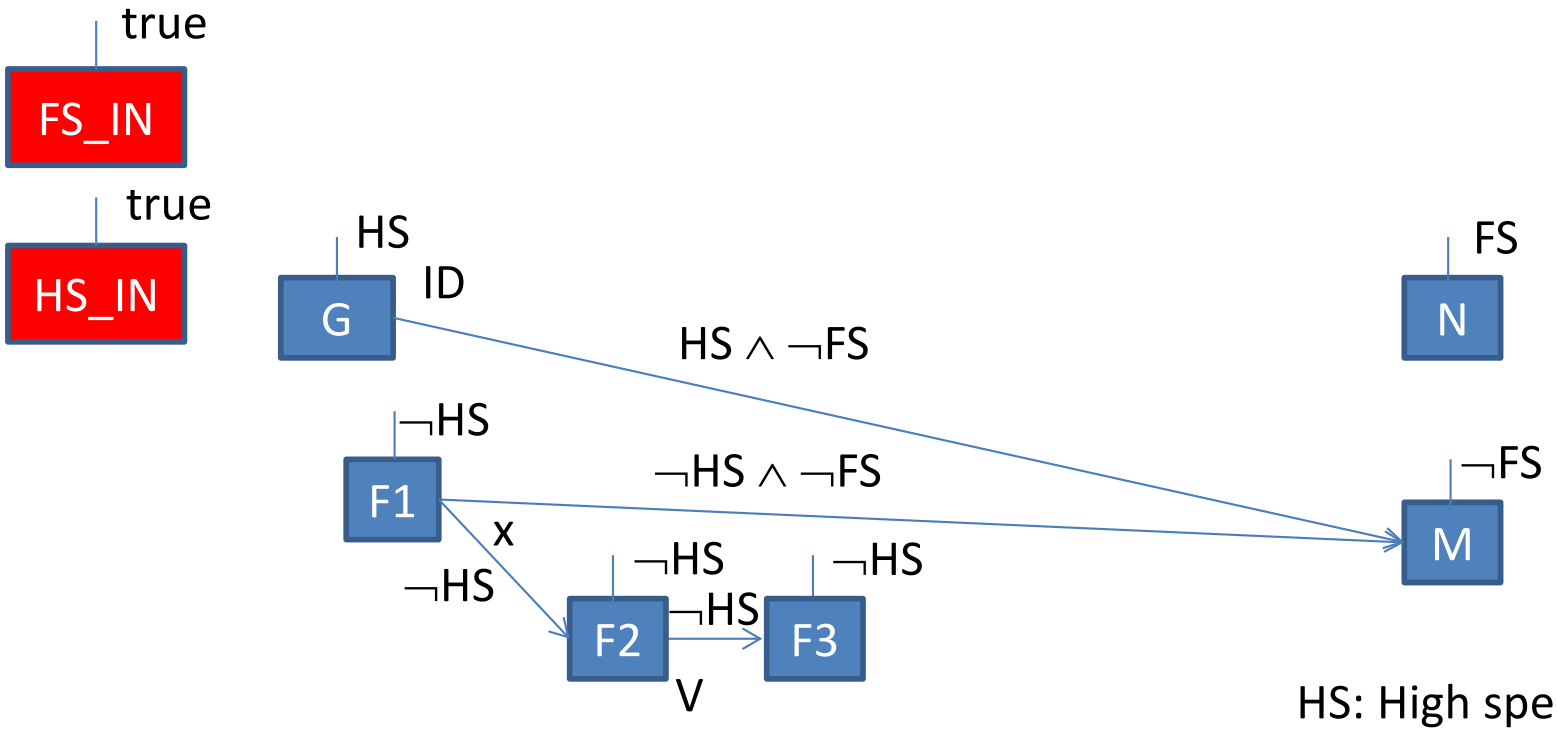
Pré-traitement : mise à plat

- Traduction en flot de données non hiérarchique



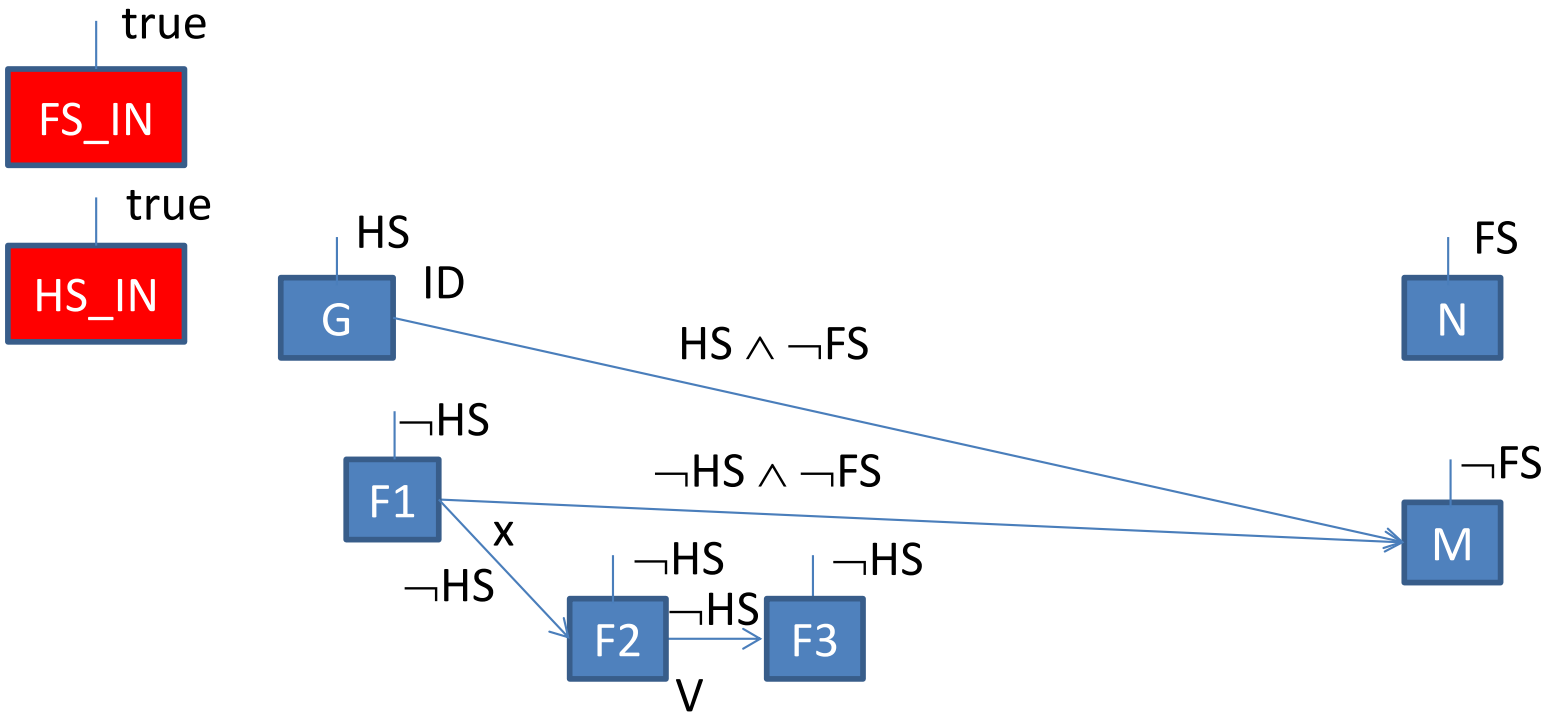
Pré-traitement : mise à plat

- Effacer les dépendances entre cycles (fby)



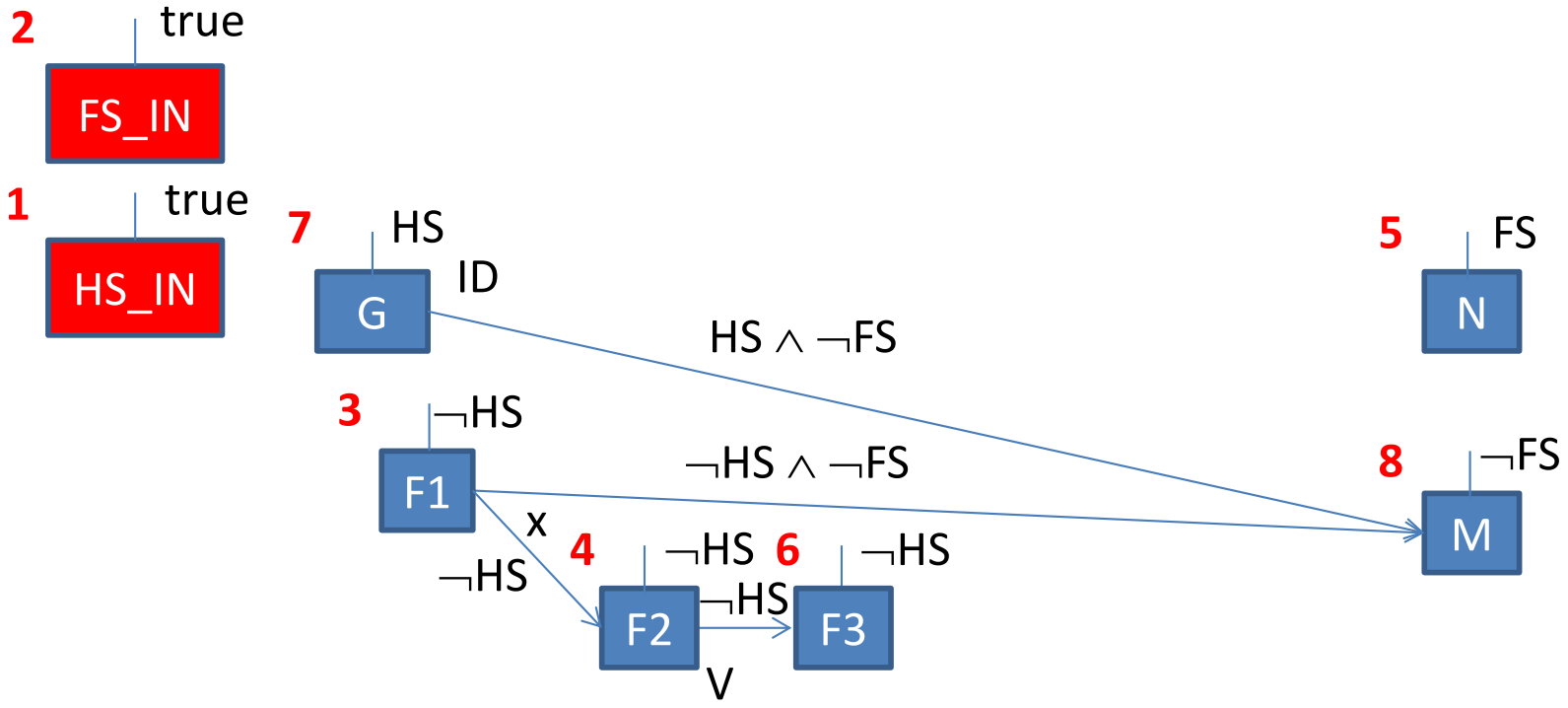
Algorithme d'ordonnancement

- Etape 1: ordonner les blocs de calcul (« list scheduling »)
  - Définition d'un ordre total compatible avec les dépendances de données (sauf celles des délais)



Algorithme d'ordonnancement

- Etape 1: ordonner les blocs de calcul (« list scheduling »)
  - Définition d'un ordre total compatible avec les dépendances de données (sauf celles des délais)



Algorithme d'ordonnancement

- Etape 2: allouer et ordonnancer les blocs 1 par 1
  - Allocation et ordonnancement optimaux pour chaque bloc pris séparément
    - Critère: Fonction de coût
      - On essaye toutes les allocations légales, on en retient exactement une parmi celles qui minimisent la fonction de coût
      - Ici: date de fin de calcul au pire cas du bloc
    - N'implique pas l'optimalité globale
      - Pas de retour en arrière (« backtracking »)
  - L'ordonnancement des communication se fait en fonction des besoins liés au placement des blocs flot de données
    - Plusieurs routes possibles => plus compliqué

Heuristique d'ordonnancement

- Etape 2: ordonnancer les blocs 1 par 1

Au début, la table de réservations est vide

time	P0	P1	P2	Bus
0				
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				

Heuristique d'ordonnancement

- Etape 2: ordonnancer les blocs 1 par 1

L'opération HS_IN ne peut être allouée que sur P0, et l'ordo optimal est à la date 0.

time	P0	P1	P2	Bus
0	HS_IN			
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				

Heuristique d'ordonnancement

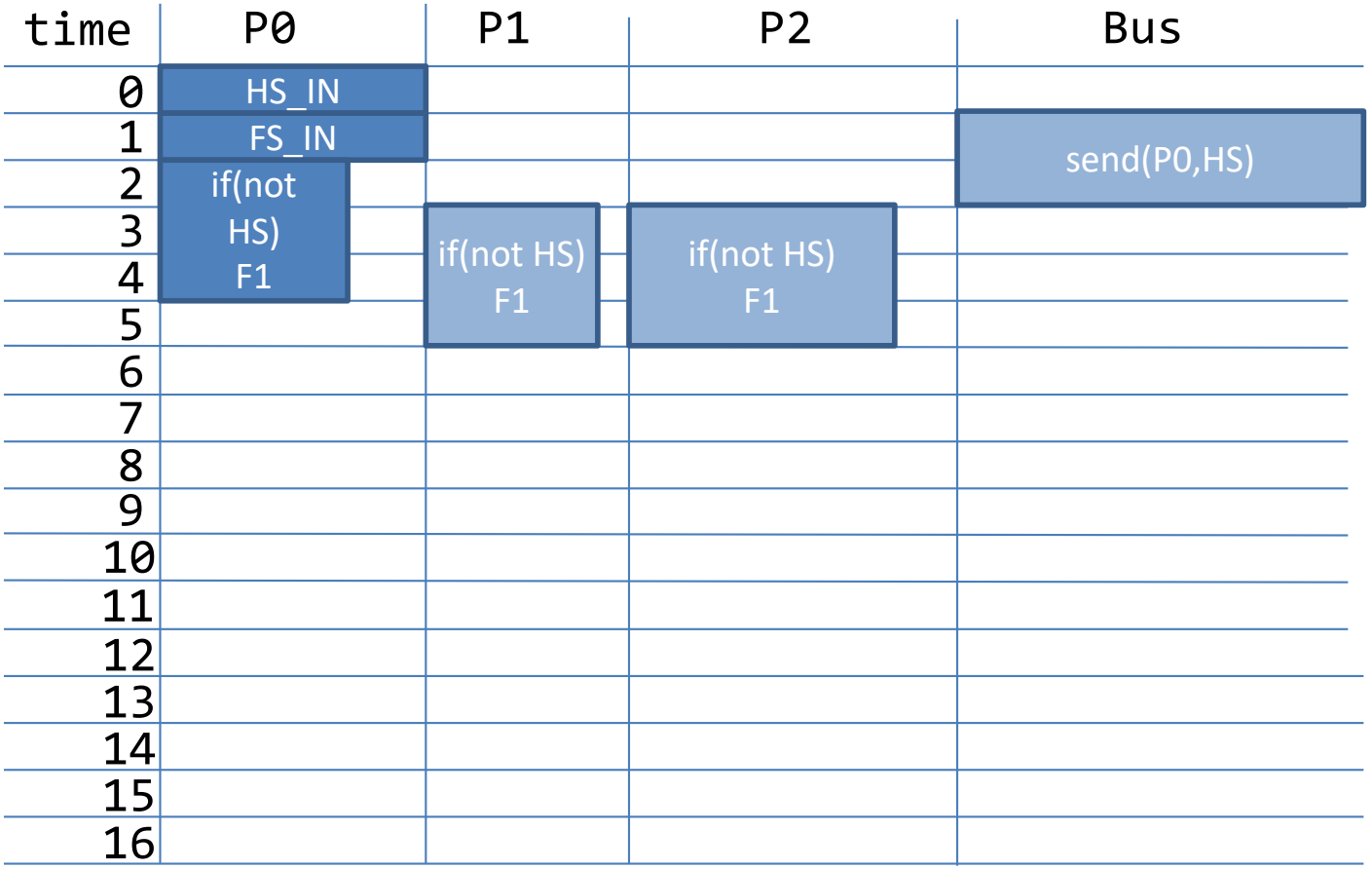
- Etape 2: ordonnancer les blocs 1 par 1

L'opération FS_IN ne peut être allouée que sur P0, et l'ordo optimal est à la date 1.

time	P0	P1	P2	Bus
0	HS_IN			
1	FS_IN			
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				

Heuristique d'ordonnancement

- Etape 2: ordonnancer les blocs 1 par 1



L'opération F1 peut être allouée sur P0, P1, et P2. Elle est allouée sur P0, car cela permet de terminer à la date 5, alors que sur P1 ou P2 sa date de fin serait 6 (à cause de la communication).

Attention: le bus est de type broadcast, donc « send » n'a pas de destinataire.

Heuristique d'ordonnancement

- Etape 2: ordonnancer les blocs 1 par 1

time	P0	P1	P2	Bus
0	HS_IN			
1	FS_IN			
2	if(not HS) F1			
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				

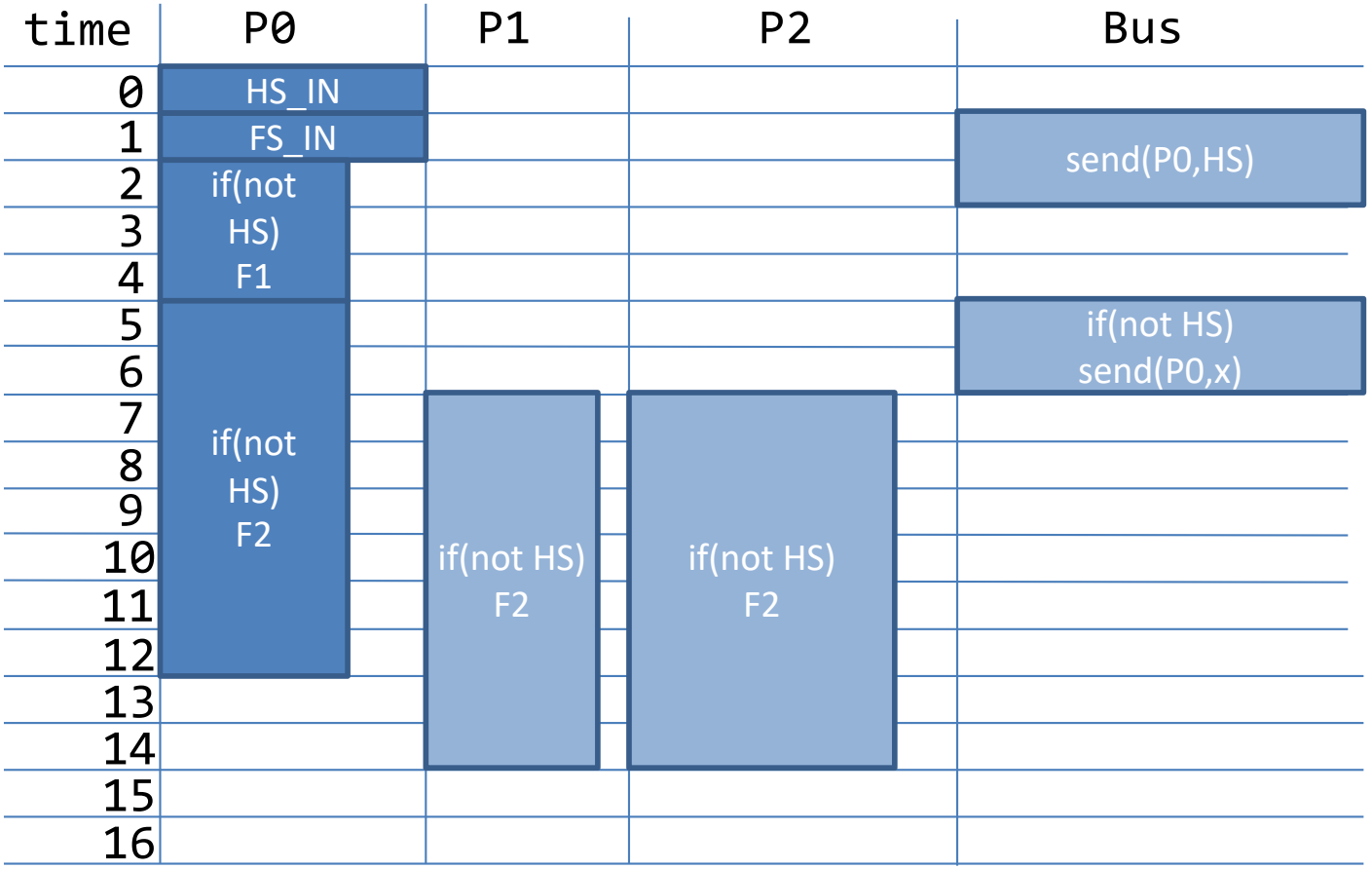
L'opération F1 peut être allouée sur P0, P1, et P2. Elle est allouée sur P0, car cela permet de terminer à la date 5, alors que sur P1 ou P2 sa date de fin serait 6 (à cause de la communication).

Attention: le bus est de type broadcast, donc « send » n'a pas de destinataire.

Heuristique d'ordonnancement

- Etape 2: ordonnancer les blocs 1 par 1

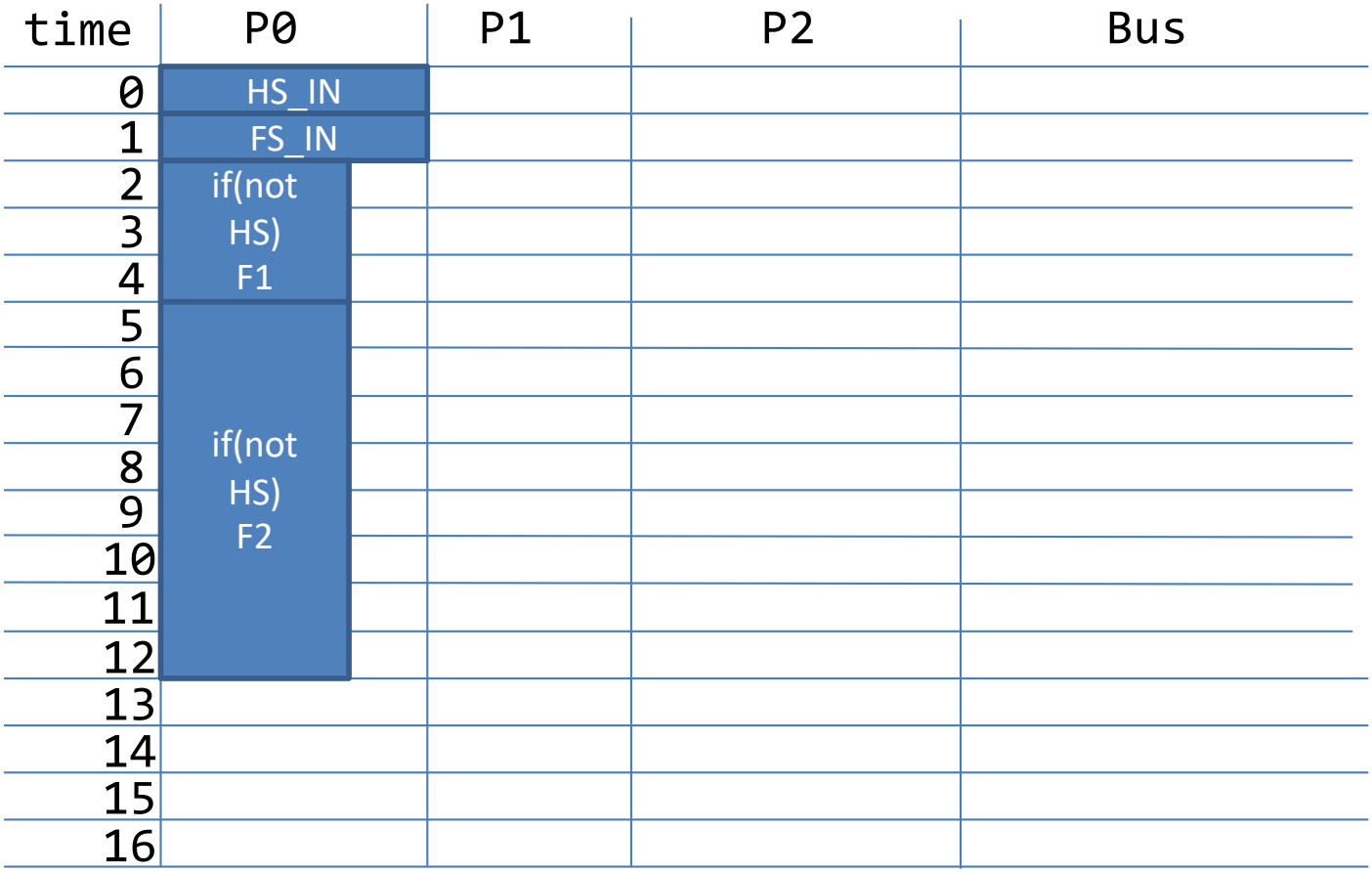
Pareil pour F2.



Heuristique d'ordonnancement

- Etape 2: ordonnancer les blocs 1 par 1

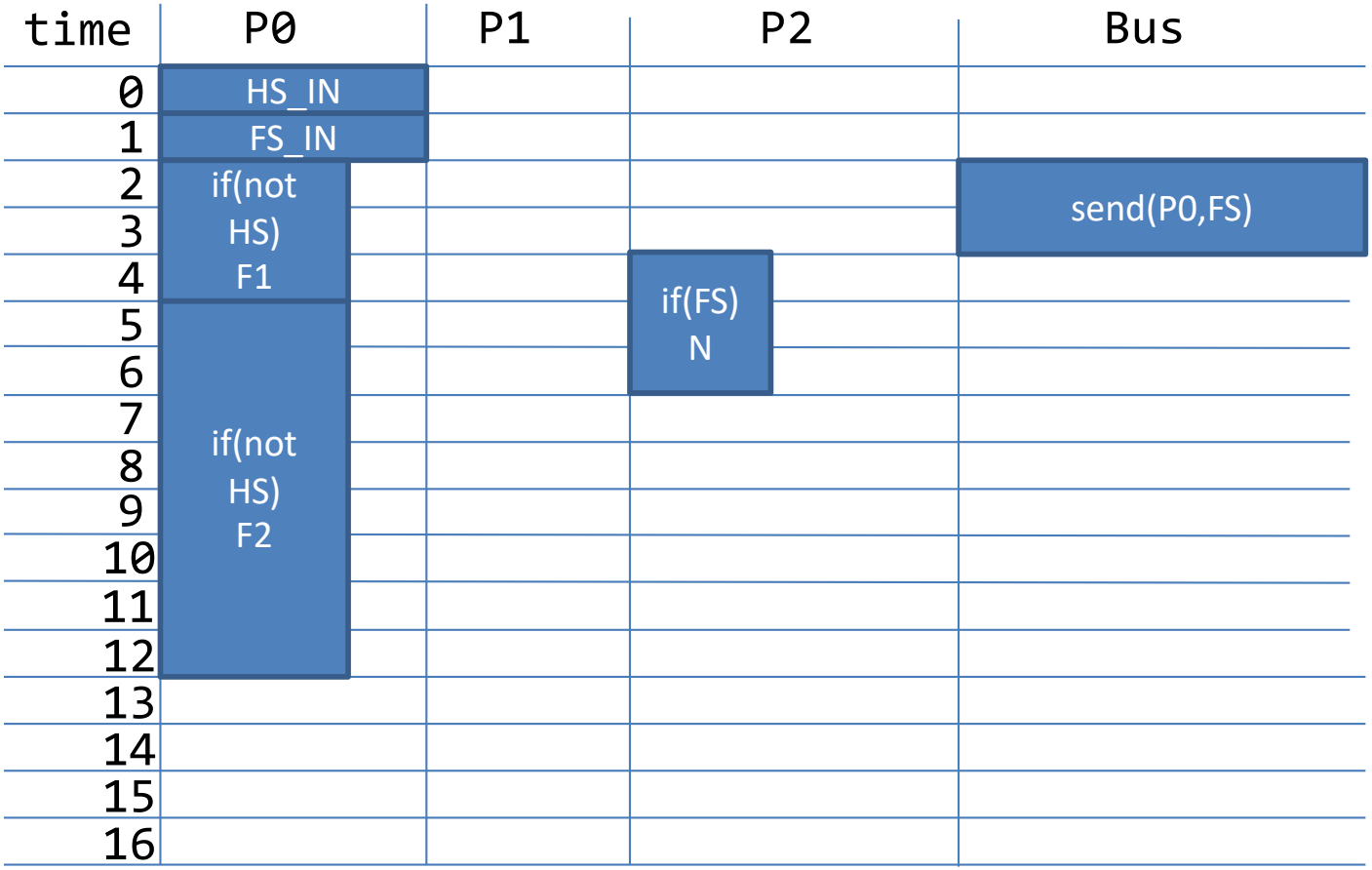
Pareil pour F2.



Heuristique d'ordonnancement

- Etape 2: ordonnancer les blocs 1 par 1

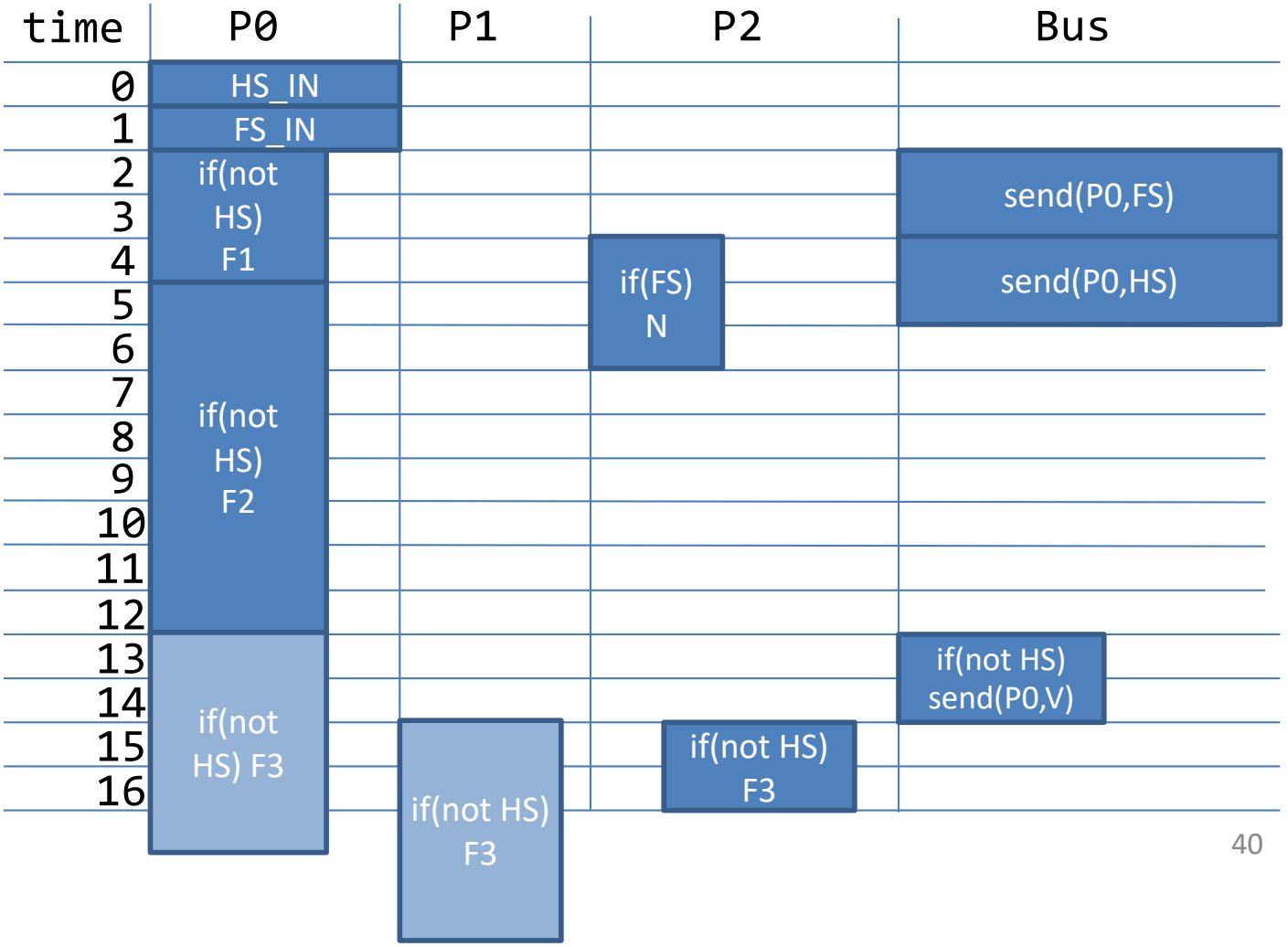
N doit être alloué sur P2 (contrainte). Cela force la transmission de FS. Les 2 opérations sont ordonnancées au plus tôt.



Heuristique d'ordonnancement

- Etape 2: ordonnancer les blocs 1 par 1

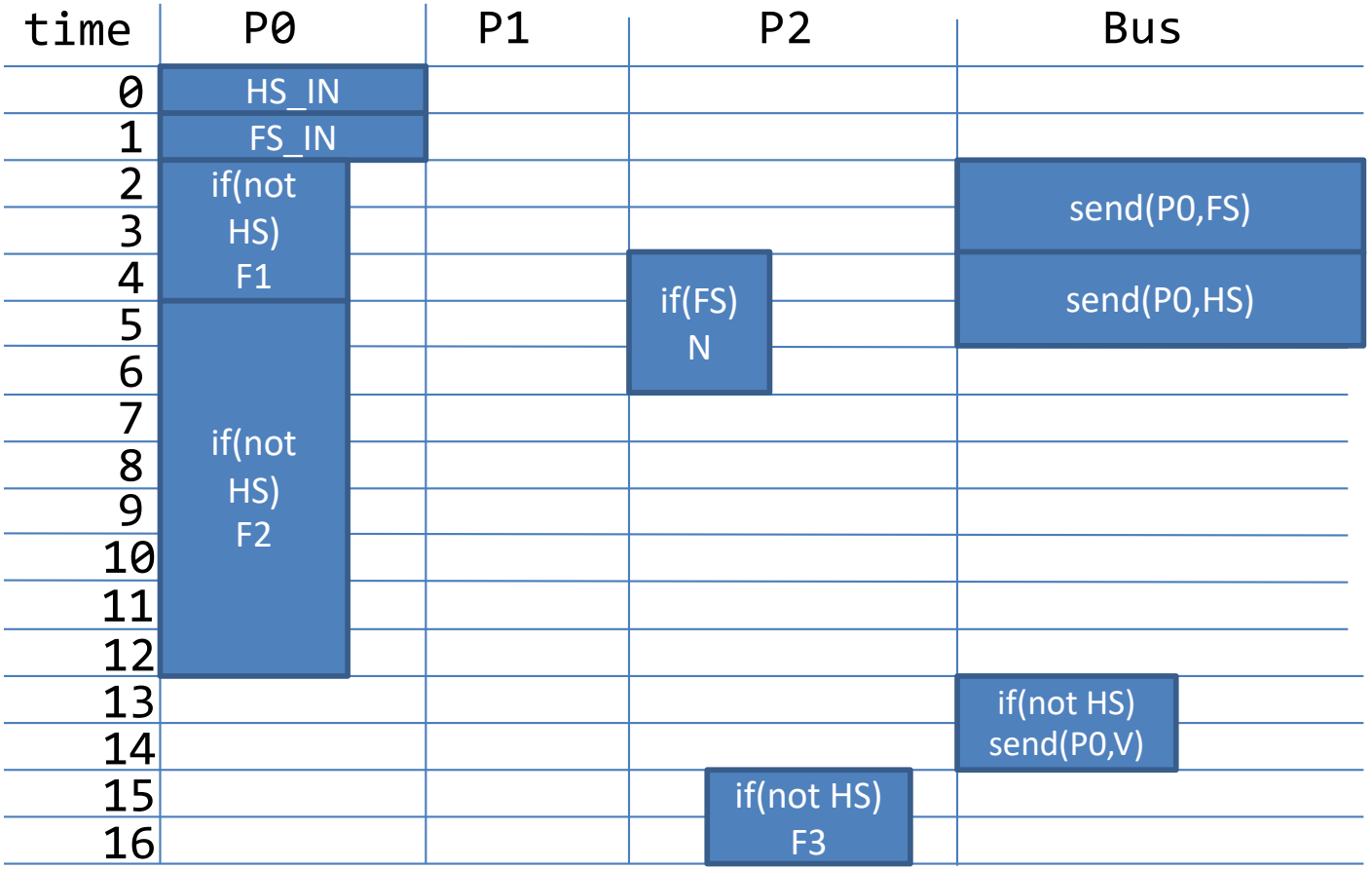
F3 peut être allouée sur P0, P1, et P2, mais elle s'exécute plus vite sur P2, ce qui compense le temps de communication.



Heuristique d'ordonnancement

- Etape 2: ordonnancer les blocs 1 par 1

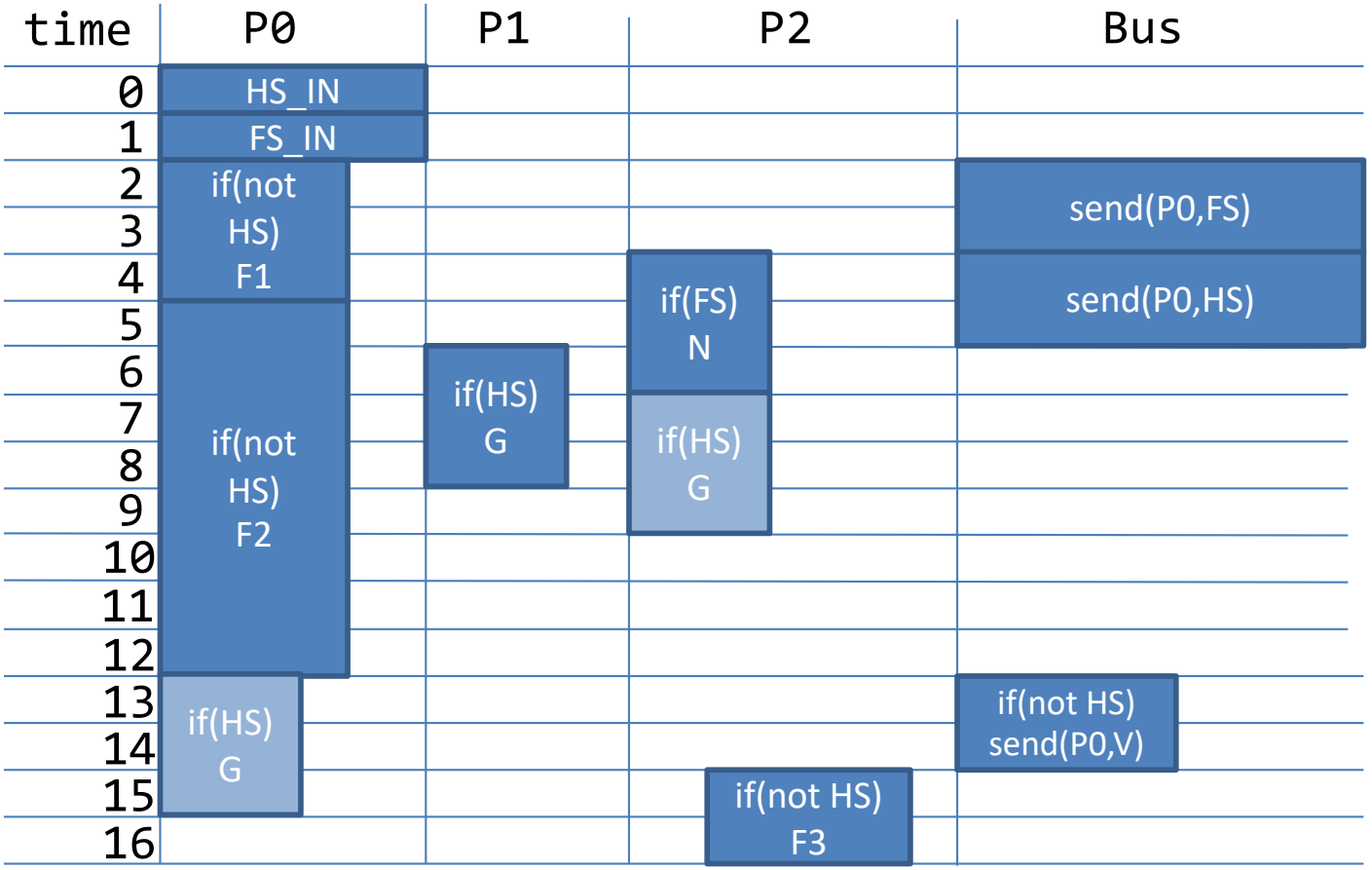
F3 peut être allouée sur P0, P1, et P2, mais elle s'exécute plus vite sur P2, ce qui compense le temps de communication.



Heuristique d'ordonnancement

- Etape 2: ordonnancer les blocs 1 par 1

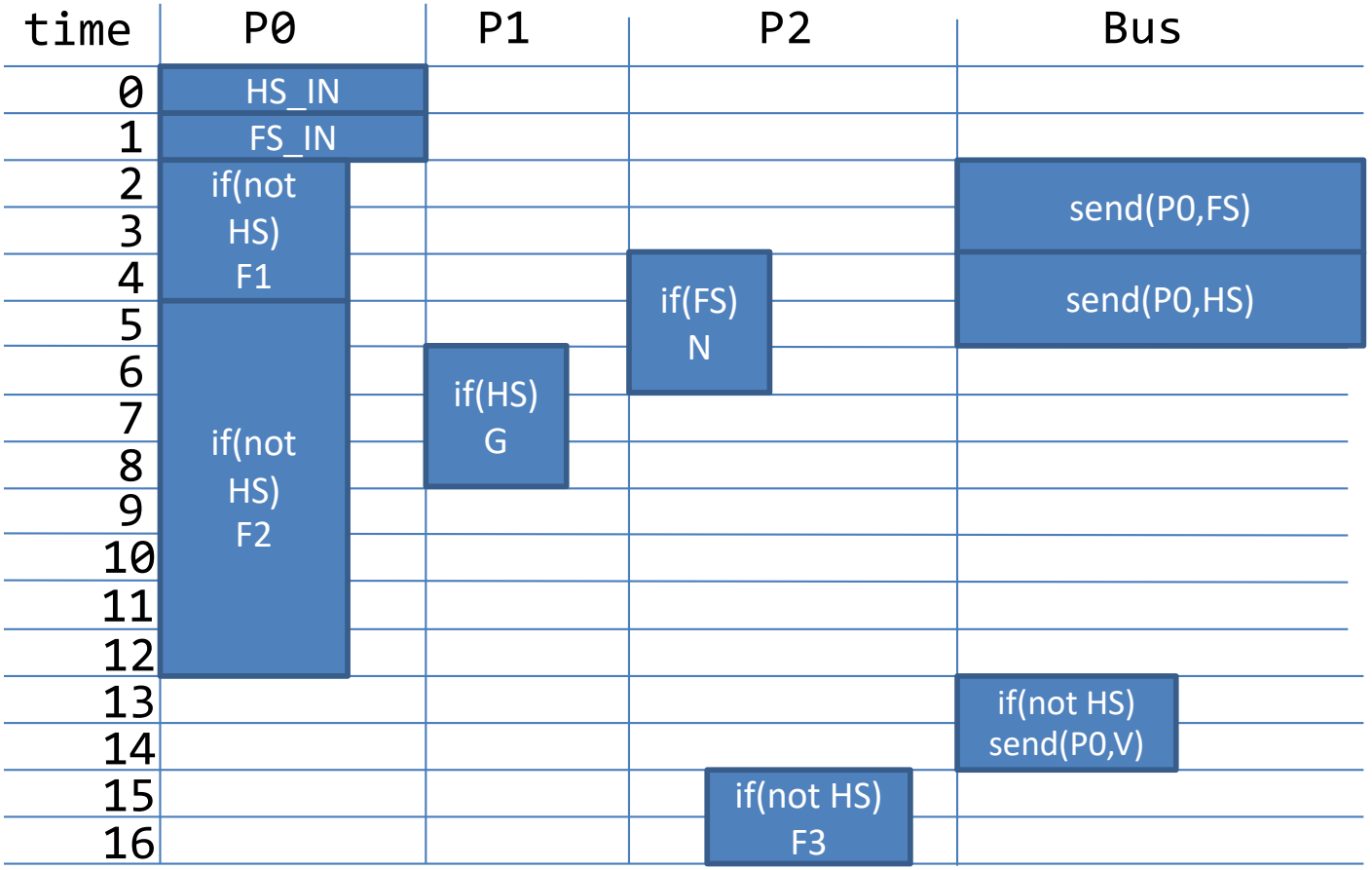
G peut être allouée sur P0, P1, et P2, mais elle se termine plus vite sur P1.



Heuristique d'ordonnancement

- Etape 2: ordonnancer les blocs 1 par 1

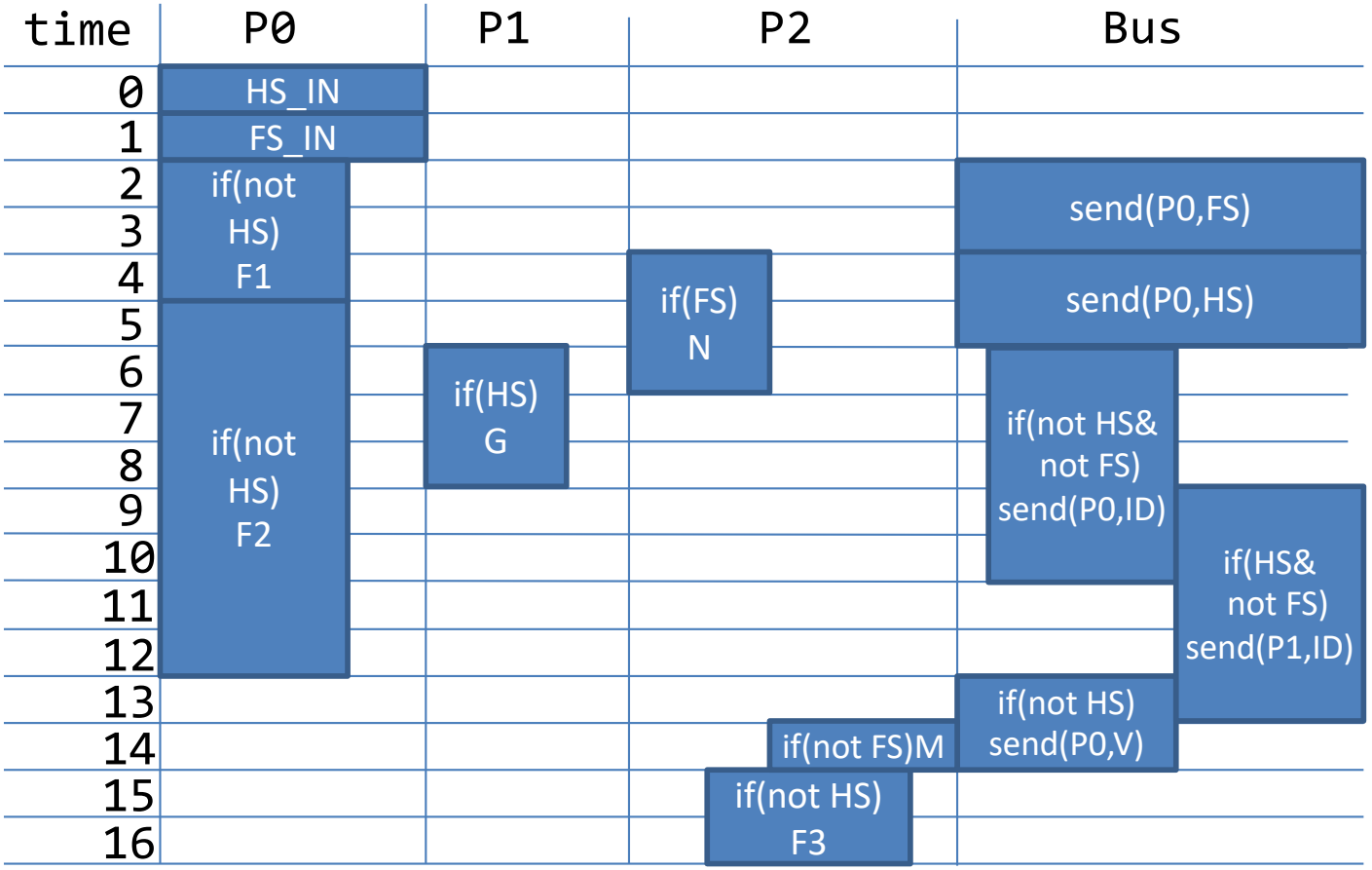
G peut être allouée sur P0, P1, et P2, mais elle se termine plus vite sur P1.



Heuristique d'ordonnancement

- Etape 2: ordonnancer les blocs 1 par 1

M doit être allouée sur P2. Sa donnée d'entrée ID peut venir soit de P0, soit de P1, en fonction de HS. Cette donnée n'est pas nécessaire quand M ne s'exécute pas.

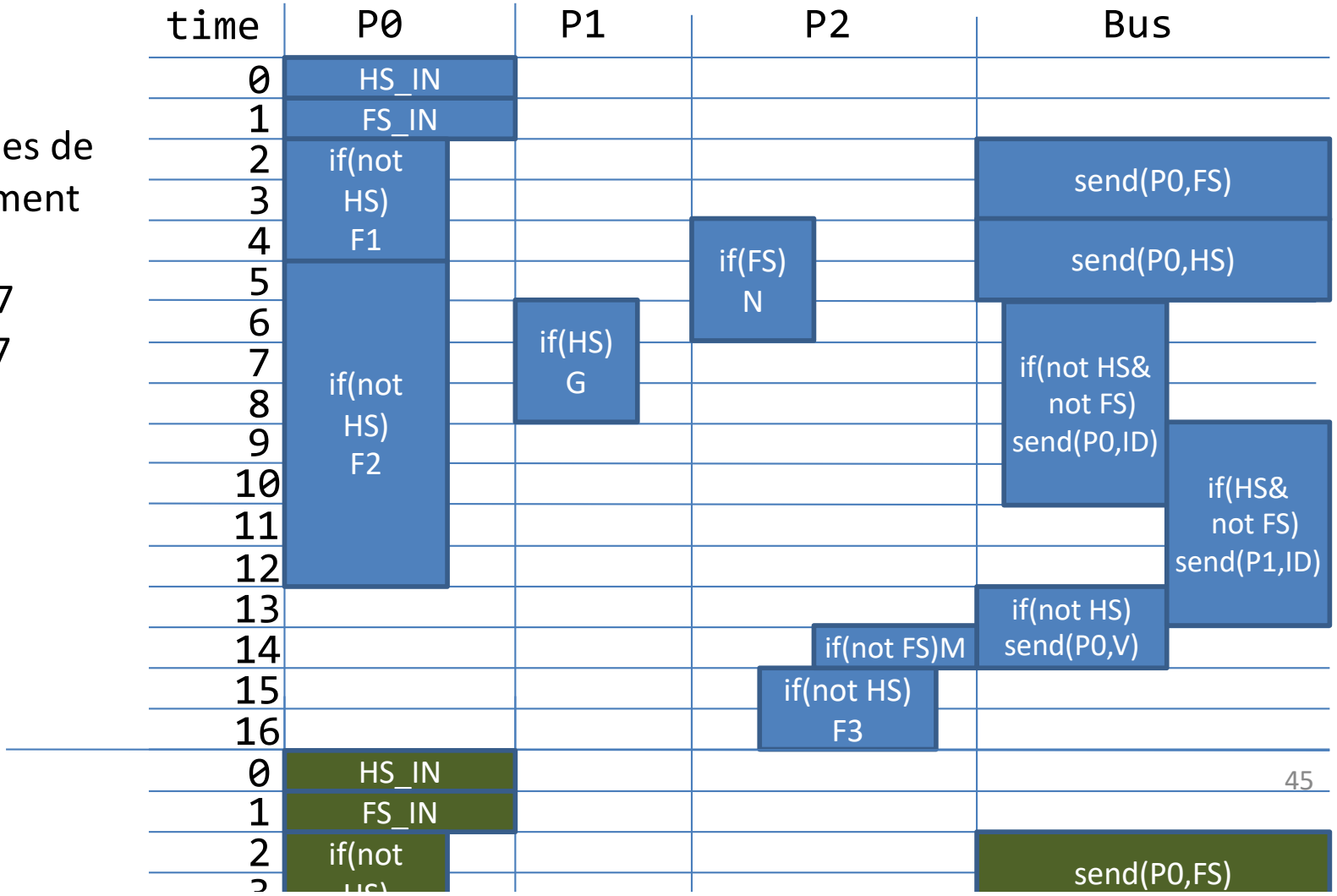


L'Allocation des communications et la gestion des conditions d'exécution est particulièrement difficile.

Heuristique d'ordonnancement

- Etape 2: résultat et exécution

Caractéristiques de l'ordonnancement généré:
- Latence: 17
- Débit: 1/17



Heuristique d'ordonnancement

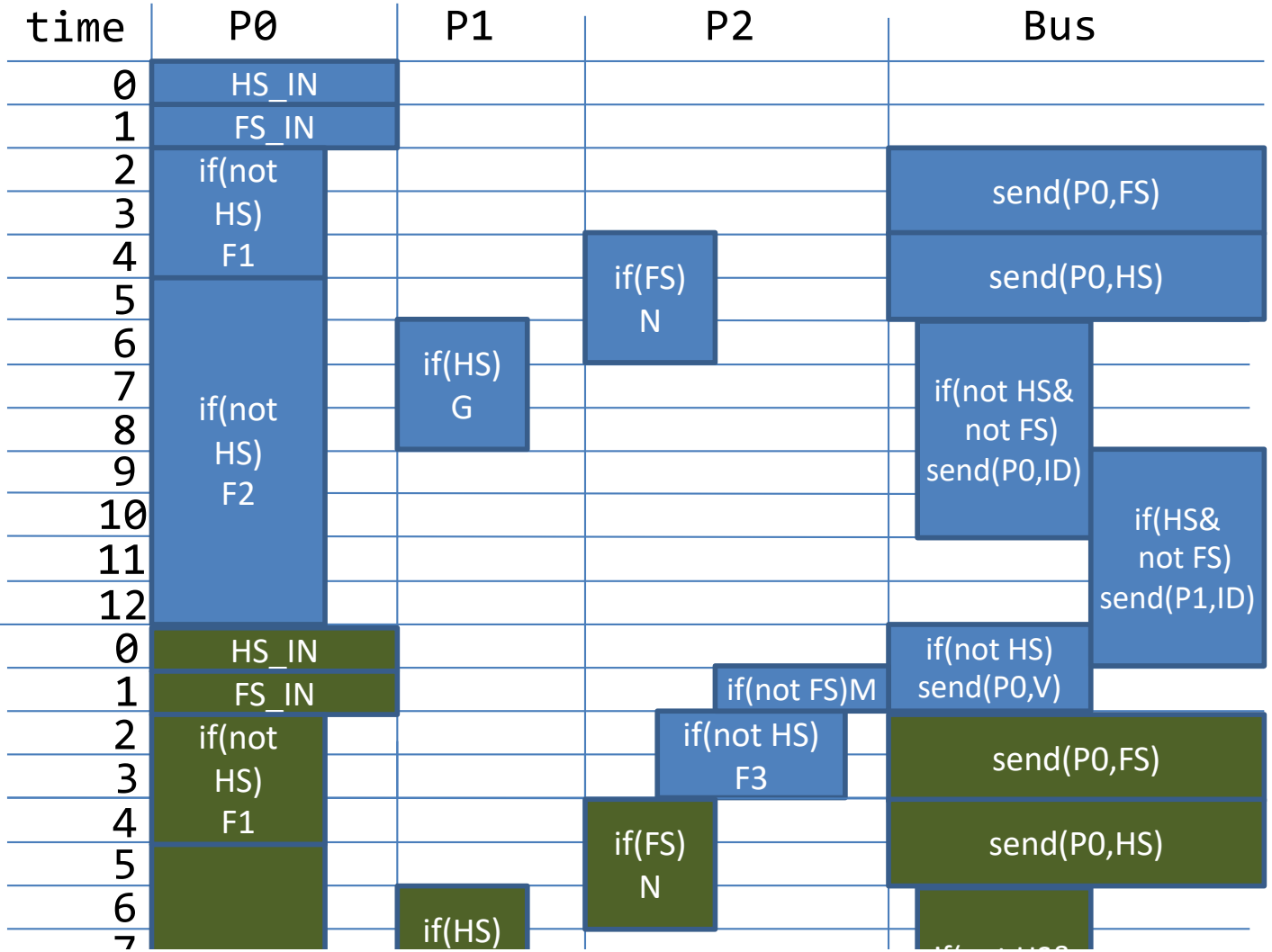
- Etape 3 (optionnelle) : pipelinage logiciel

Technique classique de compilation.

Le cycle suivant commence dès que cela est permis par les ressources.

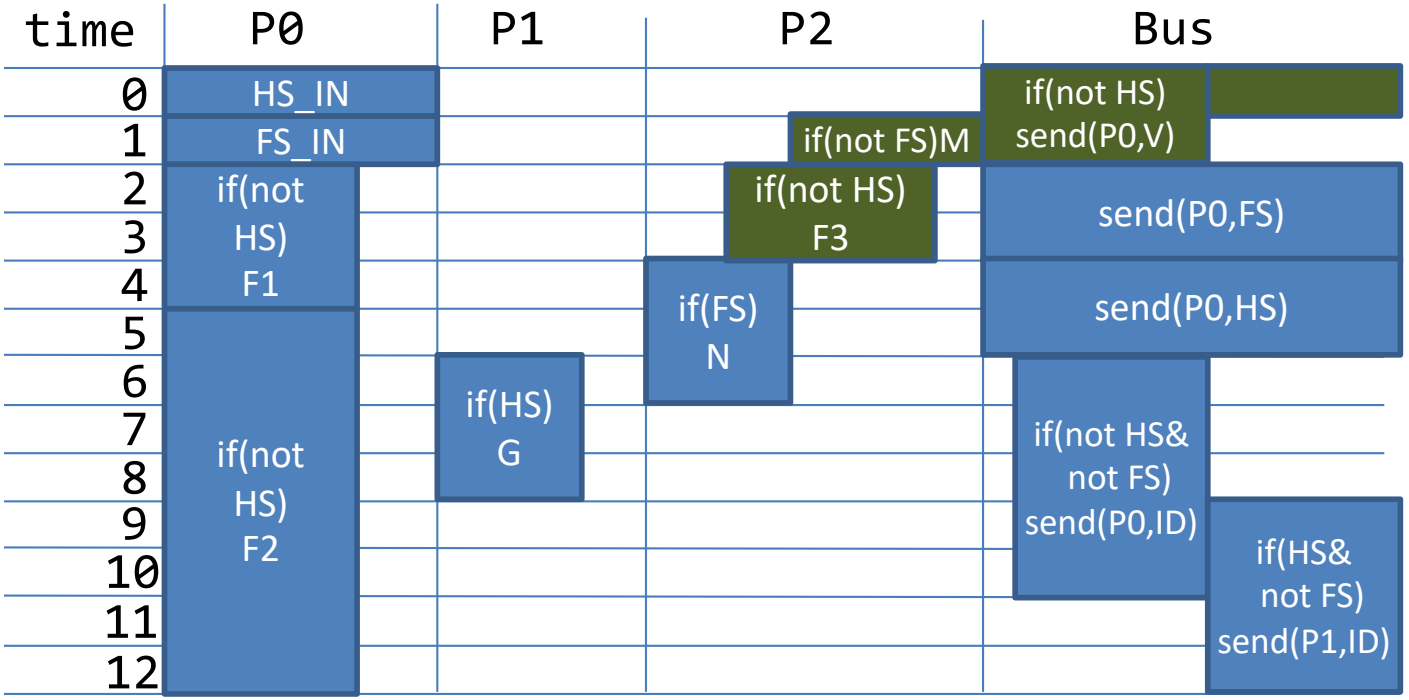
Caractéristiques du nouvel ordonnancement :

- Latence: 17
- Débit: **1/13**



Heuristique d'ordonnancement

- Etape 3 (optionnelle) : pipelinage logiciel



Technique classique de compilation.

Le cycle suivant commence dès que cela est permis par les ressources.

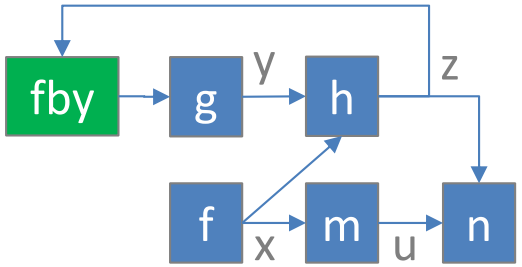
Caractéristiques du nouvel ordonnancement :

- Latence: 17
- Débit: **1/13**

En mémoire partagée

- Exemple du cours 6 – dual-core implementation

```
node main() returns ()
var x,y,z,u : int ;
let
  y  = g(123 fby z) ;
  x  = f() ;
  z  = h(x,y) ;
  u  = m(x) ;
  () = n(z,u) ;
tel
```

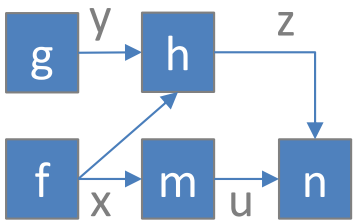


WCET(f)=WCET(g)=WCET(n)=1000
WCET(h)=WCET(m)=2000

En mémoire partagée

- Etape 1 – graphe cyclique -> DAG

```
node main() returns ()
var x,y,z,u : int ;
let
  y  = g(123 fby z) ;
  x  = f() ;
  z  = h(x,y) ;
  u  = m(x) ;
  () = n(z,u) ;
tel
```

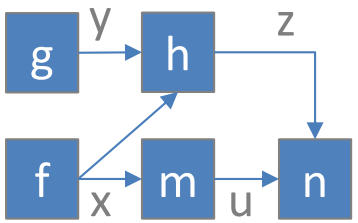


WCET(f)=WCET(g)=WCET(n)=1000
WCET(h)=WCET(m)=2000

En mémoire partagée

- Etape 2 – list scheduling

time	CPU0	CPU1
0		
1		
2		
3		
4		
5		
6		

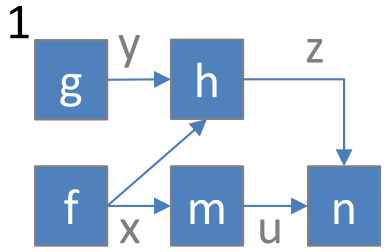


WCET(f)=WCET(g)=WCET(n)=1
WCET(h)=WCET(m)=2

En mémoire partagée

- Etape 2 – list scheduling

time	CPU0	CPU1
0	g	g
1		
2		
3		
4		
5		
6		

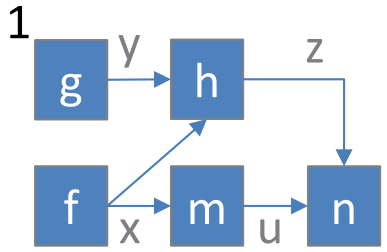


WCET(f)=WCET(g)=WCET(n)=1
WCET(h)=WCET(m)=2

En mémoire partagée

- Etape 2 – list scheduling

time	CPU0	CPU1
0	g	
1		
2		
3		
4		
5		
6		

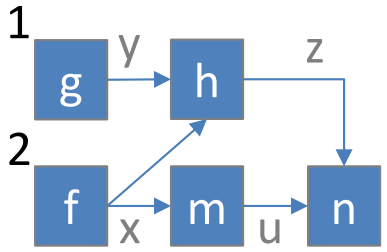


WCET(f)=WCET(g)=WCET(n)=1
WCET(h)=WCET(m)=2

En mémoire partagée

- Etape 2 – list scheduling

time	CPU0	CPU1
0	g	f
1	f	
2		
3		
4		
5		
6		

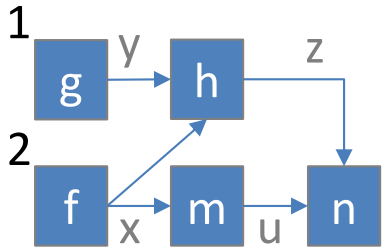


WCET(f)=WCET(g)=WCET(n)=1
WCET(h)=WCET(m)=2

En mémoire partagée

- Etape 2 – list scheduling

time	CPU0	CPU1
0	g	f
1		
2		
3		
4		
5		
6		

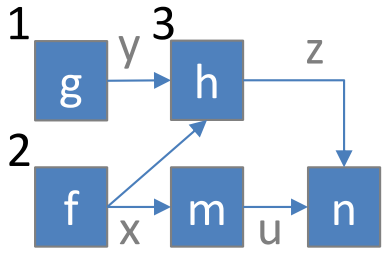


WCET(f)=WCET(g)=WCET(n)=1
WCET(h)=WCET(m)=2

En mémoire partagée

- Etape 2 – list scheduling

time	CPU0	CPU1
0	g	f
1	h	
2		
3		
4		
5		
6		

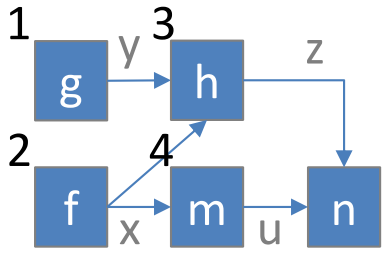


WCET(f)=WCET(g)=WCET(n)=1
WCET(h)=WCET(m)=2

En mémoire partagée

- Etape 2 – list scheduling

time	CPU0	CPU1
0	g	f
1	h	m
2		
3		
4		
5		
6		

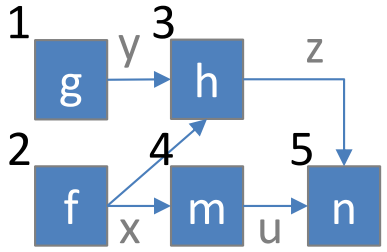


WCET(f)=WCET(g)=WCET(n)=1
WCET(h)=WCET(m)=2

En mémoire partagée

- Etape 2 – list scheduling

time	CPU0	CPU1
0	g	f
1	h	m
2		
3		n
4		
5		
6		

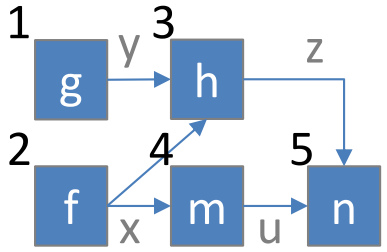


WCET(f)=WCET(g)=WCET(n)=1
WCET(h)=WCET(m)=2

En mémoire partagée

- Etape 2 – list scheduling

time	CPU0	CPU1
0	g	f
1	h	m
2		
3		n



WCET(f)=WCET(g)=WCET(n)=1
WCET(h)=WCET(m)=2

En mémoire partagée

- Synthèse de code C (y compris remise à zéro)

time	CPU0	CPU1
0	g	f
1	h	m
2		
3		n

```
void thread_cpu0(void){
  g_step(z, &y);

  WAIT_CPU  (loc_pc_1,1);
  h_step(x, y, &z);
  UPDATE_CPU(loc_pc_0,1);

  WAIT_CPU  (loc_pc_1,2);
  loc_pc_1 = -1 ;
  UPDATE_CPU(loc_pc_0,-1);
}
```

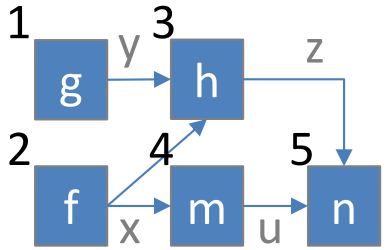
```
void thread_cpu1(void){
  f_step(&x) ;
  UPDATE_CPU(loc_pc_1,1);

  m_step(x, &u);

  WAIT_CPU  (loc_pc_0,1);
  n_step(u, z);

  UPDATE_CPU(loc_pc_1,2);
  WAIT_END  (loc_pc_0);
}
```

z

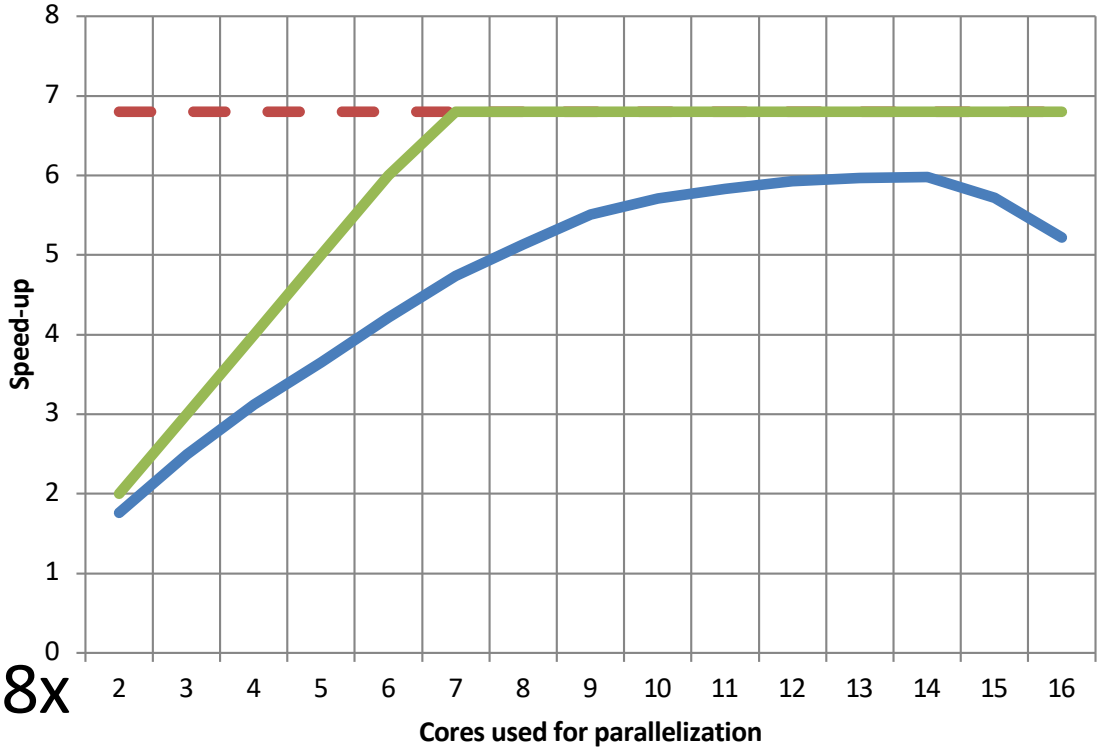


WCET(f)=WCET(g)=WCET(n)=1
WCET(h)=WCET(m)=2

```
#define WAIT_END(var) while(api_ldar(&var)!=-1)
```

Application avionique

- UC1: >5k unique nodes, >36k unique dataflow vars
- Multi-periodes
- Scalabilité
  - 8 cores: ~22s
  - 16 cores: ~43s
- Parallélisation
  - Limite théorique: 6.8x
  - 2 coeurs->1.76x, 4 coeurs->3.12x, 8 coeurs->5.13x, 12 coeurs->5.93x
  - Saturation de la bande passante à partir de 12 coeurs



Conclusion

- **La compilation de systèmes temps réel est possible**
  - Du moins pour certaines classes d'applications
  - Correction et efficacité sont assurées
  - Cela permet une approche de conception de type « trial-and-error » comme pour le logiciel non-embarqué
- Principes théoriques et algorithmiques simples
  - Indépendence, isolation, list scheduling, tables d'ordonnancement, pipelinage…
  - Mais les principes seuls ne font pas un compilateur

Préparation du TP

- Objectif unique – mise en œuvre sur bi-cœur ARM de l'ordonnanceur non-préemptif à politique EDF