Implantation sûre de systèmes contrôle/commande temps-réel

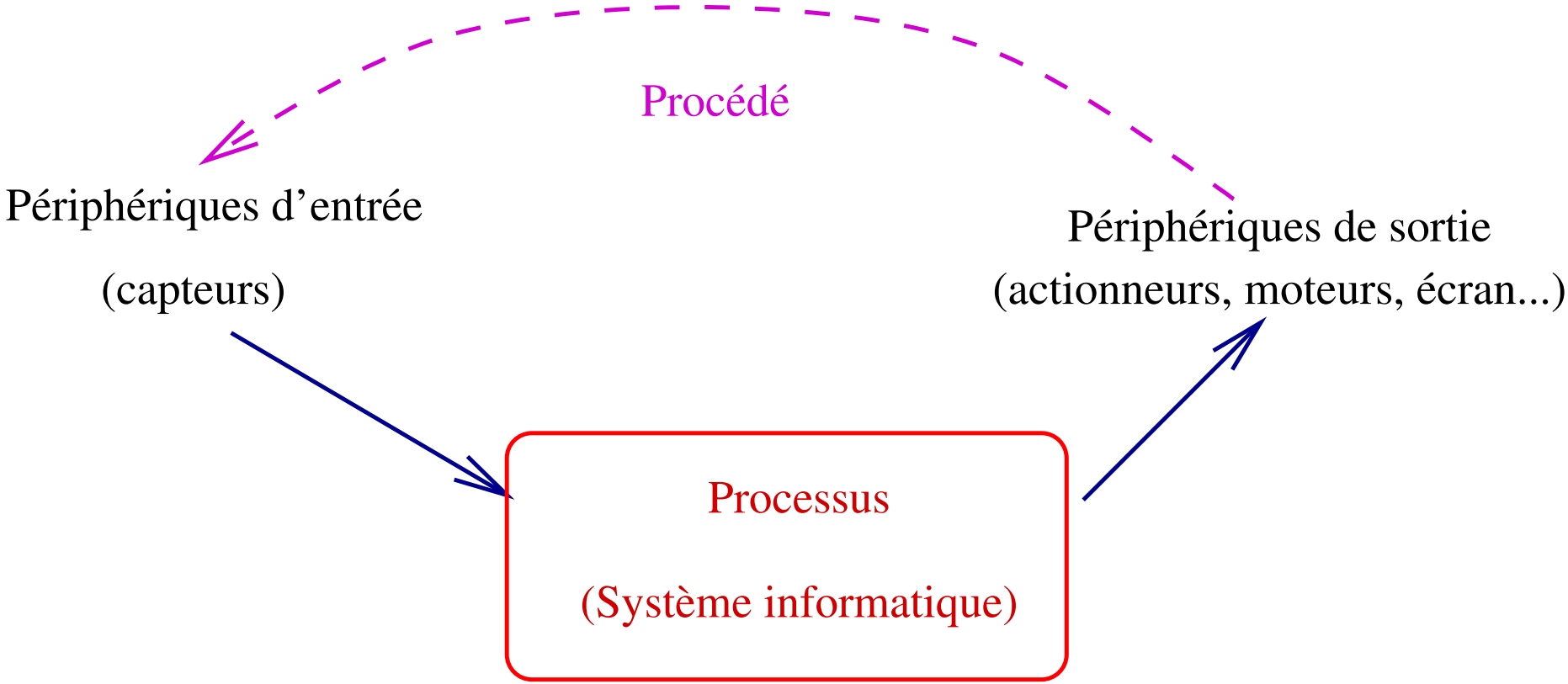
Pascal Raymond

Verimag/CNRS

3A SLE

Contrôle/commande

- Schéma général



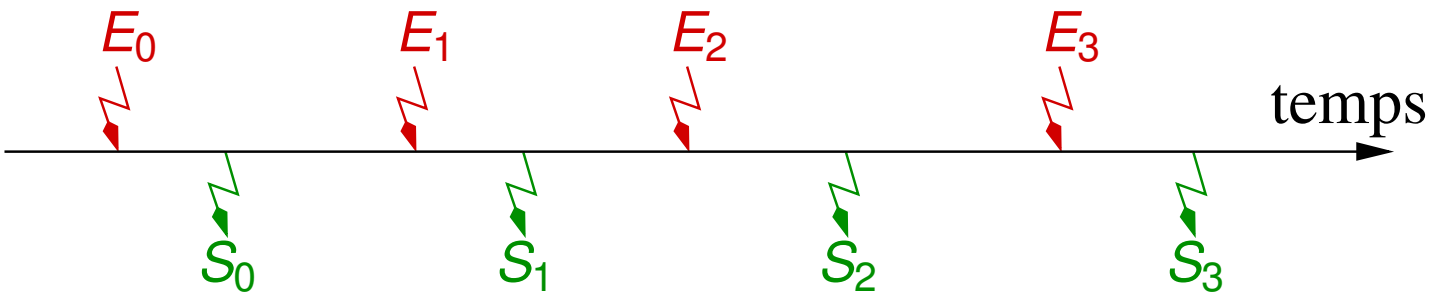
Implantation du processus

Nombreuses questions/paramètres :

- Matériel
- Logiciel de base :
  - gestion des périphériques (drivers),
  - exécutif (horloges temps-réel, allocation des ressources mémoire et calcul),
- Logiciel applicatif (la *fonction* proprement dite)

Sûreté de l'implantation ?

Fonctionnalité et temps réel



- Exécution : séquence de réactions E(ntrées)/S(orties)
- Fonctionnalité : S_t est parfaitement déterminé par $E_0, \cdots, E_t$
- Temps-réel : les E_t capturent tout changement significatif de l'environnement

Fonctionnalité et temps réel (suite)

Deux problèmes orthogonaux :

- Fonctionnalité = calculer juste, déterministe
  - Dépend de la conception, de l'implantation
- Temps-réel = calculer suffisamment vite
  - Dépend du matériel (MHz), de l'implantation ...
  - ... mais aussi de la conception

Influence de la conception sur le temps-réel

Temps de calcul non borné :

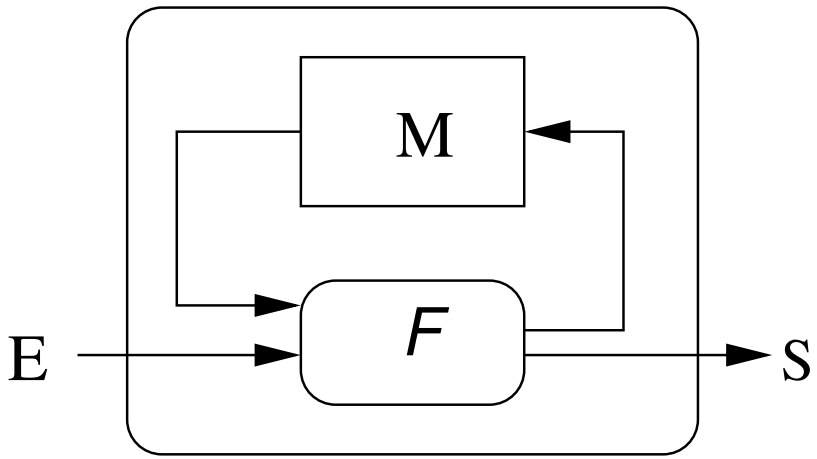
- allocation dynamique de mémoire,
- utilisation de la récursivité, de boucles ≪ while ≫
- programmation concurrente avec communication bloquante
- etc

Conclusion :

- Un système mal conçu peut être *intrinsèquement* non temps-réel
- Un système bien conçu (temps de calcul borné et évaluable) peut être temps-réel ou pas (selon la puissance de calcul)

Conception monotâche

- Quelques entrées (E) et sorties (S), réactions simples,
- identifier la mémoire nécessaire M, et sa valeur initiale M_0
- et la *fonction de transition* correcte :
  - $(S_t, M_{t+1}) = F(E_t, M_t)$
- = approche « machine à mémoire »



Implantation monotâche

- Implémenter la fonction de transition (e.g. en C)
- Écrire un programme principal où la fonction est appelée :
  - à chaque changement significatif (event-driven)
  - ou sur sur horloge périodique (échantillonnage)
    (très courant pour le contrôle/commande « hard-real-time »)

```
main() {
    M = m0;
    while(1) {
        wait_period();
        read(E);
        S, M = F(E, M);
        write(S);
    }
}
```

Exemple : chrono avec lap

Spécification :

- Entrées : start/stop, reset, lap
- Sortie : affichage

Mémoires ?

- running, bool, change sur start/stop,
- lapset, bool, change sur lap,
- counter, entier, 0 sur reset, +1 si running
- display, entier, figé si lapset, =compteur sinon

Exemple : chrono avec lap

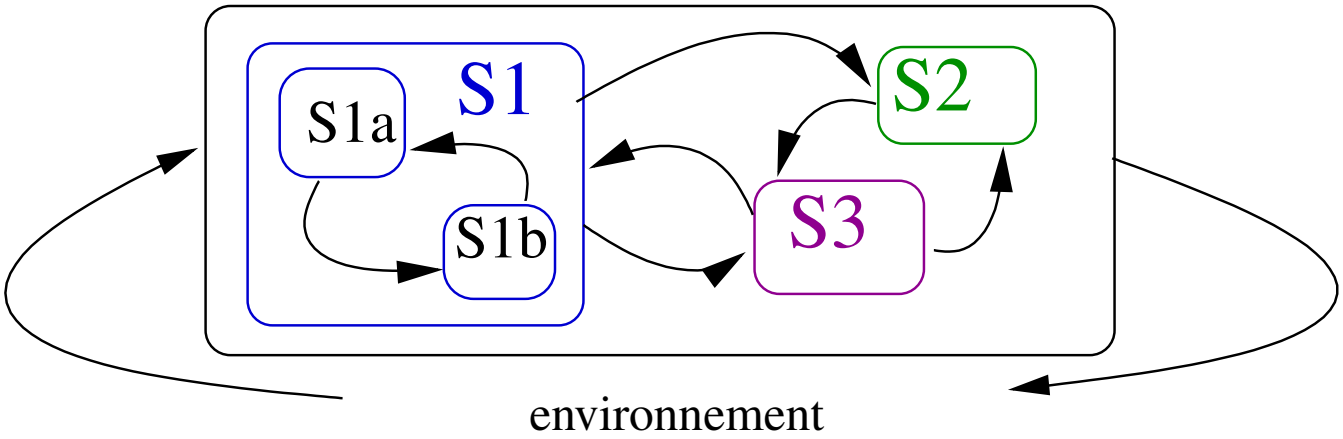
```
bool running = false;
bool lapset = false;
int counter = 0;
int display = 0;
// A appeler toutes les unités de temps
void F(ss, reset, lap) {
   running = running xor ss;
   lapset = running and (lapset xor lap);
   counter = (reset)? 0 : (running)? counter+1 : counter;
   display = (lapset)? display : counter;
}
```

Avantages du monotâche

- Fonctionnalité bien définie, déterminisme.
- Temps de calcul borné (si code simple !), évaluable pour une architecture donnée.
- Temps-réel $\Leftrightarrow$ le pire temps d'exécution $<$ période d'échantillonnage.

Conception hiérarchique et parallèle

- Gros système : beaucoup d'entrées/sorties
- Conception ≪ mono-bloc ≫ impossible
- Solution classique : découper le problème



Langages et outils

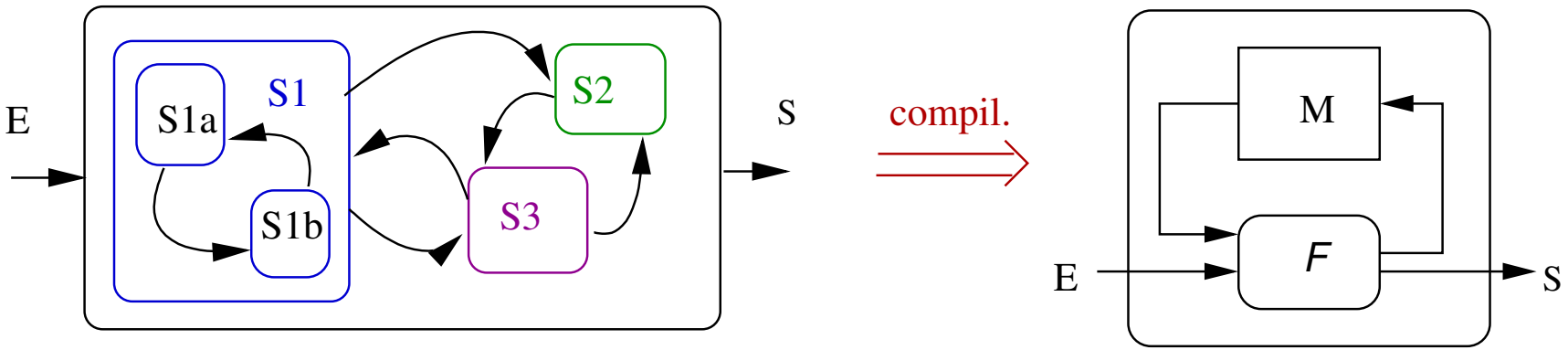
- Formalismes « Block Diagrams » pour faciliter la conception
- Plus des outils pour :
  - éditer,
  - simuler, tester, (voire vérifier)
  - générer automatiquement du code (compilateur)
- Exemples :
  - À la main : les « block diagrams » sont une spécification
  - Scade : environnement de programmation graphique, prévu dès le départ pour la génération de code
  - Simulink : à l'origine, outil de simulation, maintenant avec des générateurs de code

Problèmes sémantiques

Un block diagram ce sont :

- des systèmes et sous-systèmes concurrents et communicants,
- dont l'exécution parallèle réalise la fonctionnalité globale.
- À ce niveau, c'est du parallélisme de *description*
- sauf cas particulier, implémenter en multitâche n'apporte que des ennuis :
  - perte de performance (coût de l'exécutif)
  - problèmes sémantiques (plus profond) :
    - blocages dus aux communications,
    - courses critiques (inversion de priorité)...

Block Diagram vers monotâche



Avantages :

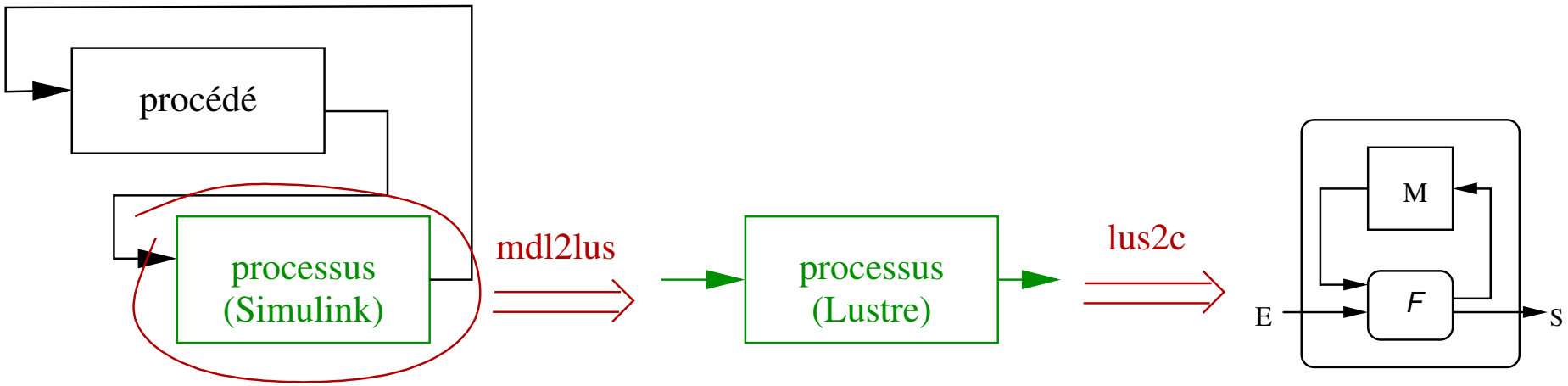
- code simple, temps d'exécution borné, évaluable
- pas de blocage/course critique : ordonnancement statique

N.B. c'est ce que font :

- le compilateur Scade
- (essentiellement) les générateurs de code pour Simulink

Model Based Design

- conception conjointe procédé/processus
- outil de référence pour automaticien (Simulink)
- génération automatique du code « contrôle/commande »via Lustre (Scade)



Intérêt du programme Lustre

Le traducteur `mdl2lus` agit comme un filtre sémantique :

- accepte les modèles avec une sémantique synchrone *claire*
- rejette des modèles « douteux », dont le fonctionnement dépend d'artefacts de simulation (e.g. ordre des écritures/lectures dans Data-Store)

Embarquer le code Lustre/c sur la brique

Choix de l'implémentation :

- monotâche périodique (pour l'instant)

Choix de plateforme :

- machine virtuelle Lego standard ?
- nxt-OSEK ?

Machine virtuelle Lego

- très ad hoc, peu représentative
- support temps-réel très limité, mais on peut implanter une tâche périodique de manière satisfaisante
- Remarques techniques :
  - c'est un système résident, avec stockage, chargement et lancement de programmes applicatifs.
  - se programme en *pseudo-c* : NXC (N(ot) e(X)actly C)
  - `lus2c` sait générer du `nxc`

Plateforme nxt-OSEK

- Vrai noyau système temps-réel embarqué, représentatif des standards industriels
  (*pour plus d'info. chercher OSEK/VDX sur internet)*
- Support temps-réel très complet, on n'a besoin que d'une toute petite partie pour le monotâche périodique
- Remarques techniques :
  - pas un système résident, mais une *librairie* système :
    - on génère un *binaire complet* en liant l'applicatif et les briques système nécessaires,
    - on charge ce binaire sur *machine nue*.
  - c'est du C très général, compilé avec `arm-elf-gcc` (cross-compiler ARM)

Environnement de travail

- Depuis les PC, tous les outils et informations sont dans :
  `/user/5/raymond/mdl2lus2osek`
- Pour simplifier :
  - ajouter une ligne dans votre `.bashrc` :
    `source /user/5/raymond/mdl2lus2osek/SETENV.sh`
  - copier (si besoin) les répertoires d'exemples chez vous.
- Documentation et slides :
  `mdl2lus2osek/current/doc`
- Exemples :
  `mdl2lus2osek/current/exemples`

Manip

- Faire tourner une appli monotâche écrite directement en C :
  - copier le répertoire modèle `c_1task_model`
  - modifier à la main le code glue (typiquement de l'affichage, pour tester)
  - compiler (make), charger (t2n), exécuter...
- Faire tourner une appli monotâche dont la fonction est écrite en simulink :
  - copier le répertoire modèle `mdl_testlight`
  - modifier à la main le code glue et le modèle simulink
  - compiler (make), charger (t2n), exécuter...

Tâches lentes/tâches rapides

- Le principe synchrone est à la base de l'implantation monotâche :
  - on conçoit avec un parallélisme de description (multitâche conceptuel)
  - ce parallélisme est *compilé* (ordonnancement statique)
- Attention : synchrone $\neq$ tout les (sous) systèmes calculent en même temps (et donc, sous-entendu) tout le temps
- *synchrone* $=$ il existe une horloge (une échelle de temps) commune à tout le monde ...
- ... mais un composant particulier peut ne calculer qu'à certains instants de cette horloge commune.

Exemple de Tâches périodiques

- une tâche urgente U, doit être effectuée toutes les $10ms$
- une tâche de fond F, doit être effectuée toutes les $40ms$ (e.g. liée à un phénomène dont la dynamique est plus lente)

Horloges :

- l'horloge temps-réel de base est le PGCD des deux (ici celle de U, soit $10ms$)
- l'horloge logique de U est "1" (calcule tout le temps)
- l'horloge logique de F est "$1/4$" (calcule une fois sur 4)

Communication entre tâches périodiques

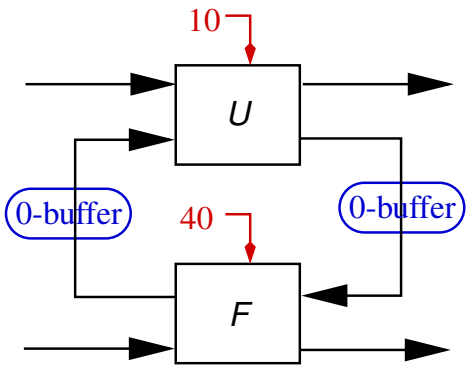
Dans le cas le plus général :

- *U* a besoin de valeurs calculées par *F*
- et *F* a besoin de valeurs calculées par *U*

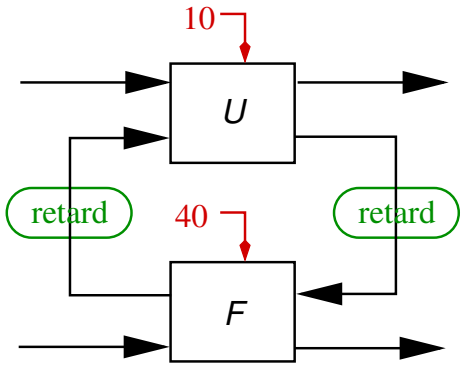
Les communications doivent être *déterministes* :

- pas de boucles combinatoires : au moins un retard logique entre *U* et *F* et/ou entre *F* et *U*
- s'il n'y a pas de retard, il faut au moins un *buffer* (zero-holder en Simulink).

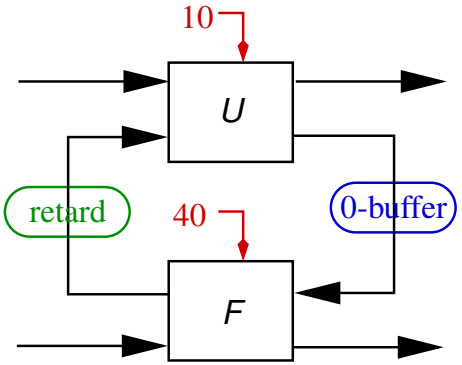
Buffers et retards : les cas possibles



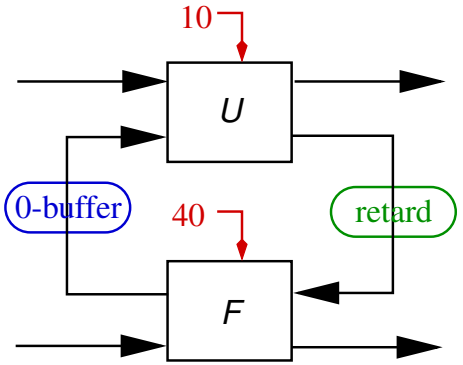
Erreur : boucle combinatoire



Toujours correct



Correct



Conceptuellement correct, refusé par Simulink

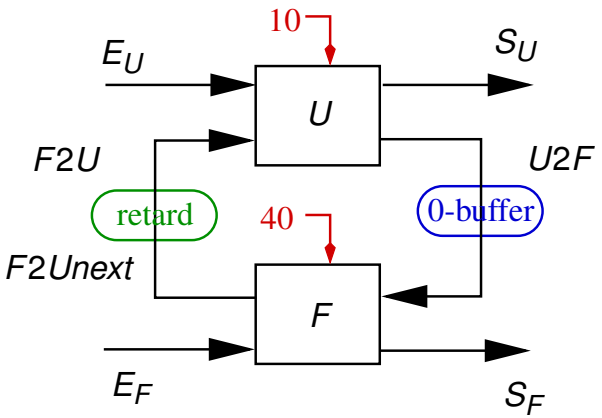
Implantation monotâche

Cas simple *harmonique*, la période longue est un multiple de la période rapide (e.g. 40 et 10) :

- un seule tâche système activée toutes les $10ms$
- calcule U tout le temps
- utilise un compteur modulo 4 pour savoir s'il faut calculer F

Conséquence : les communication sont parfaitement déterministes

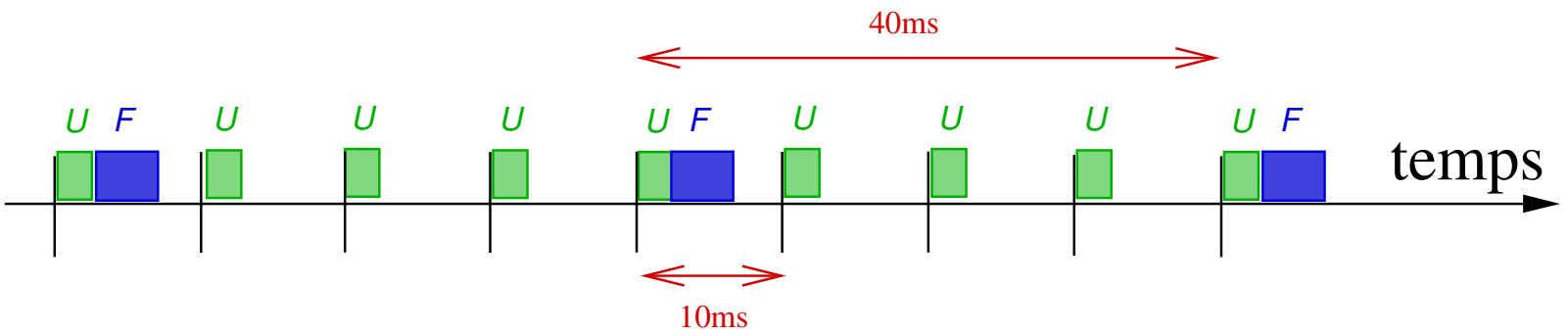
Implantation monotâche (par exemple)




```
task() {
  read(E_U);
  if (c == 0) F2U = F2Unext;
  U();
  write(S_U);
  if (c == 0) {
    write(U2F);
    read(E_F);
    F();
    write(S_F);
    write(F2Unext);
  }
  c = (c+1) mod 4;
}
```

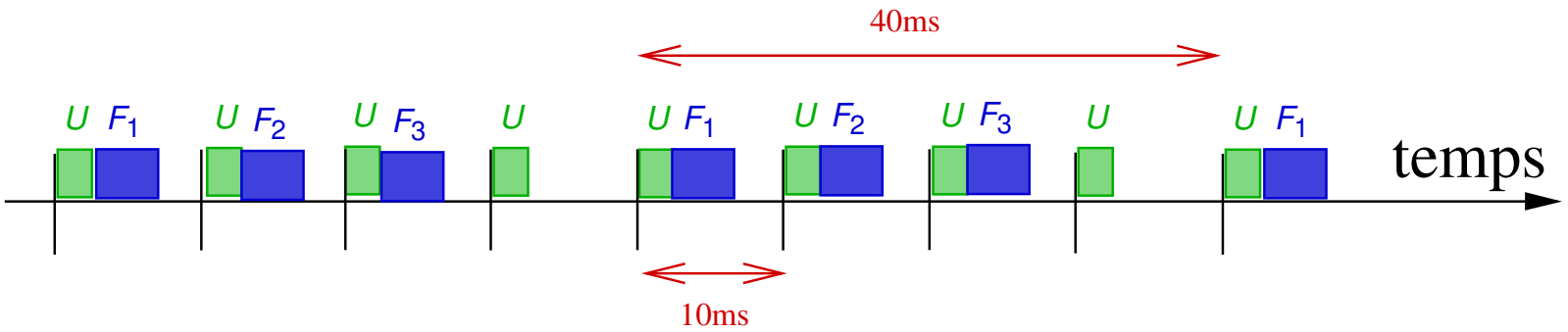
Temps d'exécution

- dans l'exemple précédent, *F* calcule moins souvent...
- ... mais doit calculer vite !
- temps-réel : WCET = WCET(U) + WCET(F) $<$ 10ms



Tâche qui prend du temps

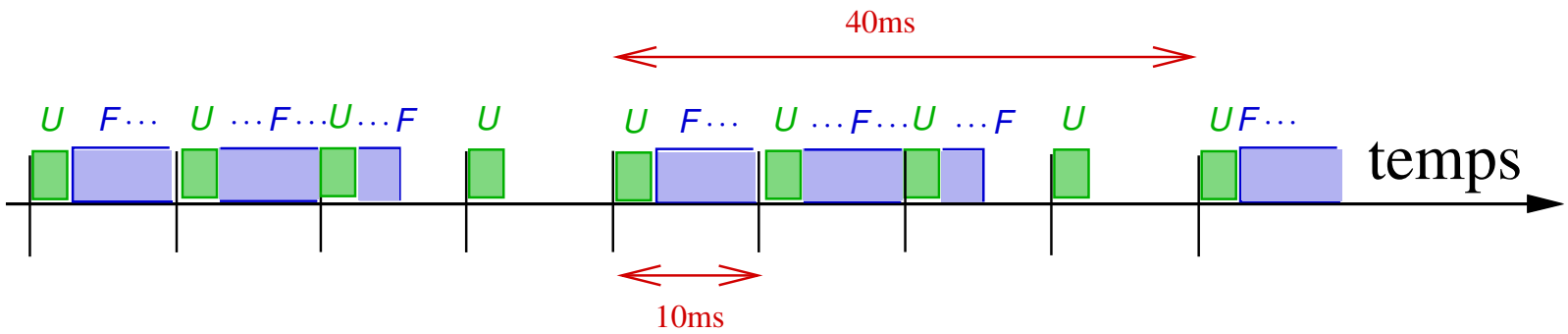
- Besoin : une tâche est activée moins souvent car elle a besoin de plus de temps pour calculer.
- Par exemple, WCET(U) = 3ms, WCET(F)=15ms
- Solution statique : découper statiquement F en 3 procédures F_1, F_2, F_3, chacune de WCET=5ms



Préemption

- Découpage statique : difficile à faire (à la main ou automatiquement)
- Découpage dynamique ? Classique dans les systèmes multitâches :
  - basé sur la *préemption*
  - ordre d'exécution déterminé par des *priorités* (on parle d'algo./politique d'ordonnancement dynamique)
- Attention : l'algo d'ordonnacement peut être assez compliqué et source d'erreur (priorités dynamiques)
- On s'intéresse à des algos *simples* et bien maîtrisés, typiquement *priorité fixe*
- Dans l'exemple : l'urgente doit être prioritaire

Préemption (exemple)



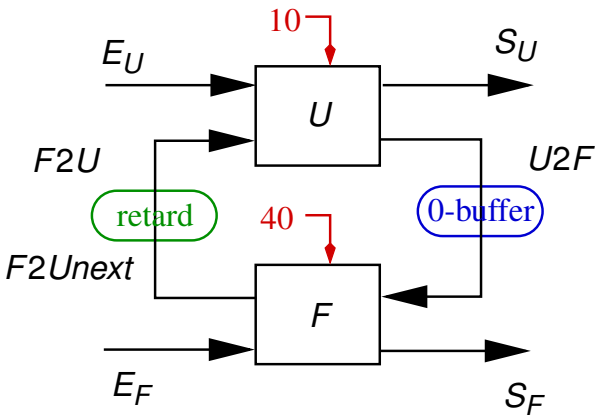
Temps réel ? Cas général un peu complexe, mais ici trivial :

- méta-période = ppcm des périodes = 40ms,
- $1 \times \text{WCET}(F) + 4 \times \text{WCET}(U) = 27 < 40\text{ms}$

Et les communications ?

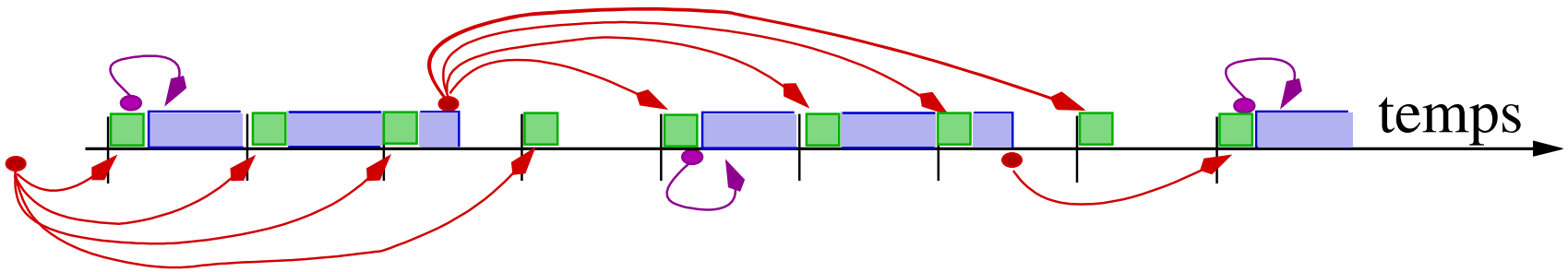
- Communication non blocante, par mémoire partagée
- Cohérence des données et déterminisme garantis par les priorités (c'est la prioritaire qui fait basculer les buffers)
- Un délai entre lente et et rapide est *obligatoire* (cf. restriction de Simulink)

Implantation multitâche (par exemple)




```
task1() {
  read(E_U);
  if (c == 0)
    F2U = F2Unext;
  U();
  write(S_U);
  if (c == 0)
    write(U2F);
  c = (c+1) mod 4;
}
```

```
task2() {
  read(E_F);
  F();
  write(S_F);
  write(F2Unext);
}
```



Implantation multitâche (par exemple)

En bref, sous les conditions suivantes :

- Cas simple, une tâche fréquente et rapide, une tâche moins fréquente et longue
- La rapide est prioritaire
- La lente doit communiquer à la rapide via un retard logique (sur l'horloge lente)
- La rapide peut communiquer directement (0-holder) vers la lente
- C'est la rapide qui se charge de basculer les buffers
- Les contraintes de WCET sont satisfaites

On a une implantation :

- multitâche préemptive, fidèle à la simulation
- sûre : déterministe, sans blocage

Les manips

- Faire tourner une appli multitâche :
  - écrite à la main en C (typiquement de l'affichage, pour tester)
    (modèle : `basic_twotasks`)
  - venant de Simulink, via mdl2lus et le compilateur Lustre
    (modèle : `harmonic`)