Le langage synchrone Esterel

Alain GIRAULT

Inria Rhône-Alpes, projet Bip

Plan de l'exposé

1. Systèmes de contrôle/commande
2. Hypothèse de synchronisme
3. Aspects fondamentaux d'Esterel
4. Compilation : automates, circuits, points d'arrêt
5. Analyse de causalité
6. Autres aspects

Programmation des systèmes de contrôle/commande

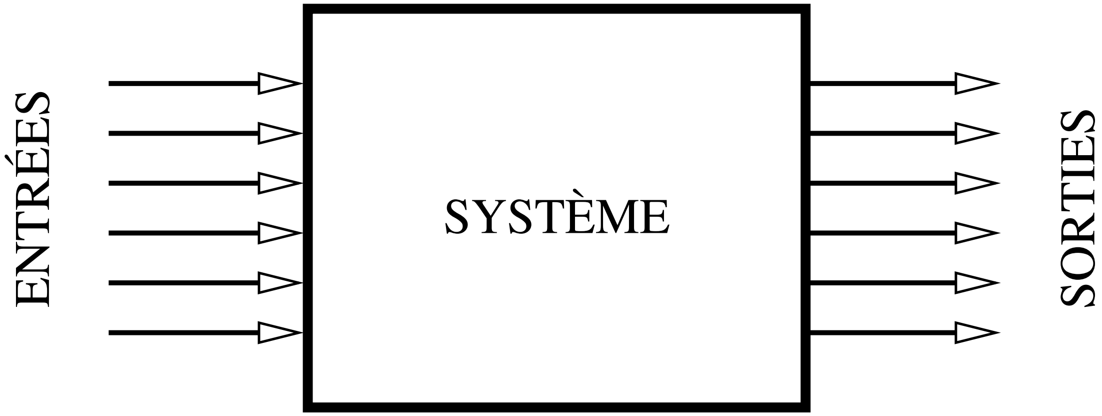
Commande = lois qui régissent l'évolution dynamique du système

- commande des actionneurs en fonctions des capteurs
- en temps continu (échantillonné)

Contrôle = lois qui régissent le comportement du système

- choix entre plusieurs lois de commande à appliquer en fonction du robot et de son environnement
- réalisation de la mission désirée et surveillance de son bon déroulement
- traitement des erreurs, des cas exceptionnels...
- en temps discret

Exemples



Contrôler le système = surveiller et agir sur son comportement

Exemples :

- véhicule automatique qui doit éviter les piétons
- robot manufacturier qui doit assembler une voiture
- robot sous-marin qui doit inspecter une installation

Contrôler le système = assurer le correct enchaînement des lois de commande

Caractéristiques de ces contrôleurs

Parallélisme : car le programme doit contrôler plusieurs automatismes en même temps

Déterminisme : car c'est plus facile de débuguer si le programme réagit toujours de la même manière aux mêmes séquences d'entrées

Temps-réel : car le système ne peut pas attendre

Sûreté de fonctionnement : car le système est critique

Approches classiques

- Systèmes câblés : hardware
- Assembleur : pour des raisons d'efficacité
- Langage classique + OS temps-réel : VxWorks, PSOS, OS9, QNX...
- Langage parallèle : ADA, OCCAM, JAVA...

Dilemme : langage de haut niveau ou langage de bas niveau ?

Critères : être parallèle, déterministe, temps-réel et sûr
➥ éviter les bugs et aller vite

Attention : les bugs temporels sont encore plus dur à trouver

Le cas de ADA

Langage de haut niveau $\Longrightarrow$ primitives temporelles de haut niveau

Mais :

- L'instruction `select` n'est pas déterministe

- L'instruction `wait 4 ; wait 5` n'est pas équivalente à `wait 9`

- La tâche qui exécute `wait 60 ; B.minute` n'envoie jamais le message `minute` en même temps que le 60e message `seconde`

L'approche synchrone

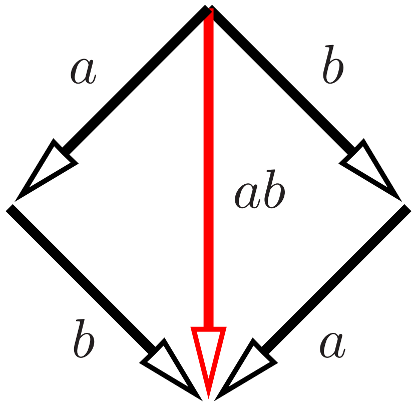
But : faciliter les raisonnements temporels

- Principe de simultanéité au lieu de l'entrelacement
  - ➥ L'indéterminisme dû à l'entrelacement disparaît
- Toutes les activités parallèles partagent la même échelle de temps qui est discrète
  - ➥ Elles peuvent être datées sur cette échelle

Ces deux principes constituent l'hypothèse de synchronisme

Simultanéité contre entrelacement

$a \| b$



L'entrelacement est proche de la mise en œuvre

- L'ordre d'exécution de a et b dépend du compilateur
- Cela oblige à se poser la question

La simultanéité est volontairement de plus haut niveau

- On ne se préoccupe pas de l'ordre d'exécution
- Le compilateur se débrouille pour que ça marche

L'échelle de temps

L'échelle discrète du temps logique est projetée sur le temps physique

Rien ne se passe entre deux instants

C'est comme si l'ordinateur était infiniment rapide

Il faut par contre vérifier les contraintes temporelles

valider l'hypothèse de synchronisme $\equiv$ majorer le temps de réaction du programme

Le temps multiforme

Il n'y a pas de notion de temps physique dans le programme

Seulement une notion d'ordre entre les événements :

➥ simultanéité ou précédence

Le temps physique est donc un événement externe

➥ Ces deux instructions sont de même nature :

1. Le train doit s'arrêter avant 10 secondes
2. Le train doit s'arrêter avant 100 mètres

Les langages synchrones

Plusieurs langages basés sur l'hypothèse de synchronisme :

- Esterel : textuel, impératif
- SyncCharts : graphique, mêmes constructions qu'Esterel
- Lustre : textuel, flots de données, fonctionnel
- Signal : textuel, flots de données, relationnel
- Argos : graphique, automates
- StateCharts : graphique, automates

Aspects fondamentaux d'Esterel

Langage synchrone impératif, avec une syntaxe textuelle

Un programme est composé de plusieurs modules en parallèle

Les entrées et sorties sont des signaux (purs ou valués)

La communication entre les modules est la diffusion synchrone

Les types simples sont préexistants (entier, réel...)

Les types complexes peuvent être importés depuis C

Primitives de base

`nothing`	ne fait rien et termine instantanément
`pause`	termine au début de l'instant suivant
`halt`	attend indéfiniment
`X := e`	affectation
`call P (X1, X2) (e1, e2)`	appel de procédure
`emit S`	émission de signal par diffusion synchrone
`p ; q`	mise en séquence instantanée
`p || q`	mise en parallèle
`loop p end`	boucle indéfinie
`repeat e times p end`	boucle définie
`if e then p else q end`	branchement sur test de l'expression
`signal S in p end`	déclaration d'un signal local

Primitives temporelles

`present S then p else q end`	branchement sur présence du signal `S`
`await d`	attend la prochaine occurrence de `d`
`abort p when d`	`d` est la garde de `p`
`weak abort p when d`	`p` a droit à ses dernières volontés
`loop p each d`	boucle temporelle
`every d do p end`	attend le 1er `d` avant de démarrer
`suspend p when d`	`d` est la garde de `p`

Un délai `d` est construit à partir de signaux

Exemples : `3 Second`, `[Second and not Meter]`, `immediate S`...

Exceptions : `trap` et `exit`

`trap T in p end`	la portée est lexicale
`exit T`	termine le `trap T` correspondant

Beaucoup de constructions temporelles peuvent être définies à l'aide de `trap`/`exit`

Exemple : `weak abort p when S`

```
trap T in
    p;
    exit T
  ||
    await S;
    exit T
end
```

Avantages du synchronisme

L'instruction `await 4 Second ; await 5 Second` est équivalente à `await 9 Second`

Le module qui exécute `every 60 Second do emit Minute` émet le signal `Minute` en même temps que la 60^{e} occurrence du signal `Seconde`

On peut aussi bien écrire `abort Train when 10 Second` que `abort Train when 100 Meter`

L'instruction

```
present A then
  Something
end present;
||
present B then
  SomethingElse
end present;
```

réagit instantanément à l'arrivée de `A`, de `B`, ou des deux en même temps

Un exemple : ABRO

Spec : Emettre le signal de sortie `O` dès l'occurrence des deux signaux d'entrée `A` et `B`. Recommencer à chaque occurrence du signal d'entrée `R`.

Un exemple : ABRO

Spec : Emettre le signal de sortie `O` dès l'occurrence des deux signaux d'entrée `A` et `B`. Recommencer à chaque occurrence du signal d'entrée `R`.

```
module ABRO:
input A, B, R;
output O;




end module
```

Un exemple : ABRO

Spec : Emettre le signal de sortie O dès l'occurrence des deux signaux d'entrée A et B. Recommencer à chaque occurrence du signal d'entrée R.

```
module ABRO:
input A, B, R;
output O;

  [ await A || await B ];


end module
```

Un exemple : ABRO

Spec : Emettre le signal de sortie `O` dès l'occurrence des deux signaux d'entrée `A` et `B`. Recommencer à chaque occurrence du signal d'entrée `R`.

```
module ABRO:
input A, B, R;
output O;

  [ await A || await B ];
  emit O;

end module
```

Un exemple : ABRO

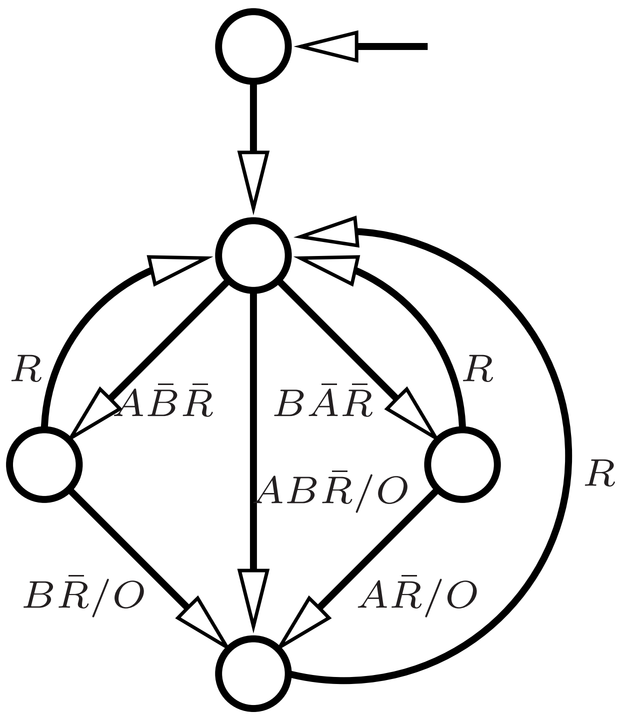
Spec : Emettre le signal de sortie `O` dès l’occurrence des deux signaux d’entrée `A` et `B`. Recommencer à chaque occurrence du signal d’entrée `R`.

```
module ABRO:
input A, B, R;
output O;
loop
  [ await A || await B ];
  emit O;
each R
end module
```

Un exemple : ABRO

Spec : Emettre le signal de sortie O dès l'occurrence des deux signaux d'entrée A et B. Recommencer à chaque occurrence du signal d'entrée R.

```
module ABRO:
input A, B, R;
output O;
loop
  [ await A || await B ];
  emit O;
each R
end module
```



L'entrainement matinal

```
module Runner:
input Second, Meter, Lap, Morning, Step;
output ...; % not given here
every Morning do
  abort
    loop
      abort RunSlowly when 15 Second;
      abort
        every Step do
          Jump || Breathe
        end every
      when 100 Meter;
      FullSpeed
    each Lap
  when 2 Lap
end every
end module
```

Si on a le coeur fragile

```
trap HeartAttack in
  abort
    loop
      abort RunSlowly when 15 Seconds;
      abort
        every Step do
          Jump || Breathe || CheckHeart
        end every
      when 100 Meter;
      FullSpeed
    each Lap
  when 2 Lap
handle HeartAttack do
  GoToHospital
end trap
```

CheckHeart doit contenir l'instruction `exit HearAttack`

Jump et *Breathe* sont alors préemptés

Compilation en automates

Le compilateur simule l'évolution des instructions de contrôle en fonction des événements courants

- Instructions de contrôle : `if`, `present`, `trap`, `exit`, `await`...

- Événements courants : occurrence des signaux d'entrée et locaux, exécution des `exit`...

L'ensemble des configurations des instructions de contrôle est fini
➥ On obtient un automate d'états fini déterministe

Le format OC

C'est le format de description des automates

Module OC = automate d'états fini déterministe étendu avec une table d'actions

➥ Afin de manipuler les objets non discrets : variables entières et réelles...

Chaque état contient un graphe orienté sans cycle d'actions (DAG)

- Les noeuds unaires sont des actions
- Les noeuds binaires sont des tests d'input ou de variable
- Les feuilles sont des gotos

Tous les objets internes sont tabulés pour une représentation compacte et efficace : types, variables, signaux, états...

```
module: ABRO

types: 0
end:

constants: 0
end:

functions: 0
end:

procedures: 0
end:

signals: 4
0: input: A 1 pure: bool: 0 %name: A% %previous: first:% %lc: 3 7 0%
1: input: B 3 pure: bool: 1 %name: B% %previous: 0% %lc: 3 10 0%
2: input: R 5 pure: bool: 2 %name: R% %previous: 1% %lc: 3 13 0%
3: output: O 6 pure: %name: O% %previous: 2% %lc: 5 8 0%
end:
```

Exemple de module OC

```
variables: 3
0: $0 %sigbool: 0% %lc: 3 7 0%
1: $0 %sigbool: 1% %lc: 3 10 0%
2: $0 %sigbool: 2% %lc: 3 13 0%
end:

actions: 7
0: call: $0 (0) (@$0)
1: present: 0 %lc: 3 7 0%
2: call: $0 (1) (@$0)
3: present: 1 %lc: 3 10 0%
4: call: $0 (2) (@$0)
5: present: 2 %lc: 3 13 0%
6: output: 3 %lc: 5 8 0%
end:

states: 6
startpoint: 1
sink: 0
calls: 27
```

Exemple de module OC

```
0: <0>

1: <2>

2: 5 (<2>) () 3 (1 (6 <3>) () <4>) () 1 (<5>) () <2>
%awaited: 0 1 2%

3: 5 (<2>) () <3>
%awaited: 2%

4: 5 (<2>) () 1 (6 <3>) () <4>
%awaited: 0 2%

5: 5 (<2>) () 3 (6 <3>) () <5>
%awaited: 1 2%

end:

endmodule:
```

Compilation en circuits

Circuits séquentiels : partie combinatoire plus des registres

Circuit séquentiel $\equiv$ automate implicite

➥ Les registres codent les états

Automate à N états $\equiv$ circuit séquentiel à log(N) registres

Permet d'éviter l'explosion combinatoire

Le compilateur génère des portes logiques et des fils pour les instructions ; seul le `pause` génère un registre

Le format SC

C'est le format de description des circuits séquentiels

Module SC = circuit séquentiel étendu avec une table d'actions

La partie combinatoire contient des fils spéciaux qui, quand le front montant arrive, déclenchent une action de la table

Là encore tout est tabulé, et les tables sont communes avec les autres formats

Exemple de module SC

```
module: ABRO

signals: 4
0: input: A 0 pure: bool: 0 %previous: first:% %lc: 3 7 0%
1: input: B 1 pure: bool: 1 %previous: 0% %lc: 3 10 0%
2: input: R 2 pure: bool: 2 %previous: 1% %lc: 3 13 0%
3: output: O 3 pure: %previous: 2% %lc: 5 8 0%
end:

variables: 3
0: $0 %sigbool: 0% %lc: 3 7 0%
1: $0 %sigbool: 1% %lc: 3 10 0%
2: $0 %sigbool: 2% %lc: 3 13 0%
end:

actions: 4
0: present: 0 %lc: 3 7 0%
1: present: 1 %lc: 3 10 0%
2: present: 2 %lc: 3 13 0%
3: output: 3 %lc: 5 8 0%
```

Exemple de module SC


```
end:

nets: 56
registers: 4
signals: 4

-- Signal 0
0: A__I_ in: 0 ift: 0
%inst: 0%

1: PRESENT_S0_0_
0
%inst: 0%
%sip: 0 0 3 7 0%

2: TRACE_S0_ sig: 0
1
%inst: 0%
%lc: 3 7 0%
-- Signal 1
```

Exemple de module SC

```
3: B__I_ in: 1 ift: 1
%inst: 0%

4: PRESENT_S1_0_
3
%inst: 0%
%sip: 1 0 3 10 0%

5: TRACE_S1_ sig: 1
4
%inst: 0%
%lc: 3 10 0%
-- Signal 2

6: R__I_ in: 2 ift: 2
%inst: 0%

...
```

Compilation en points d'arrêt

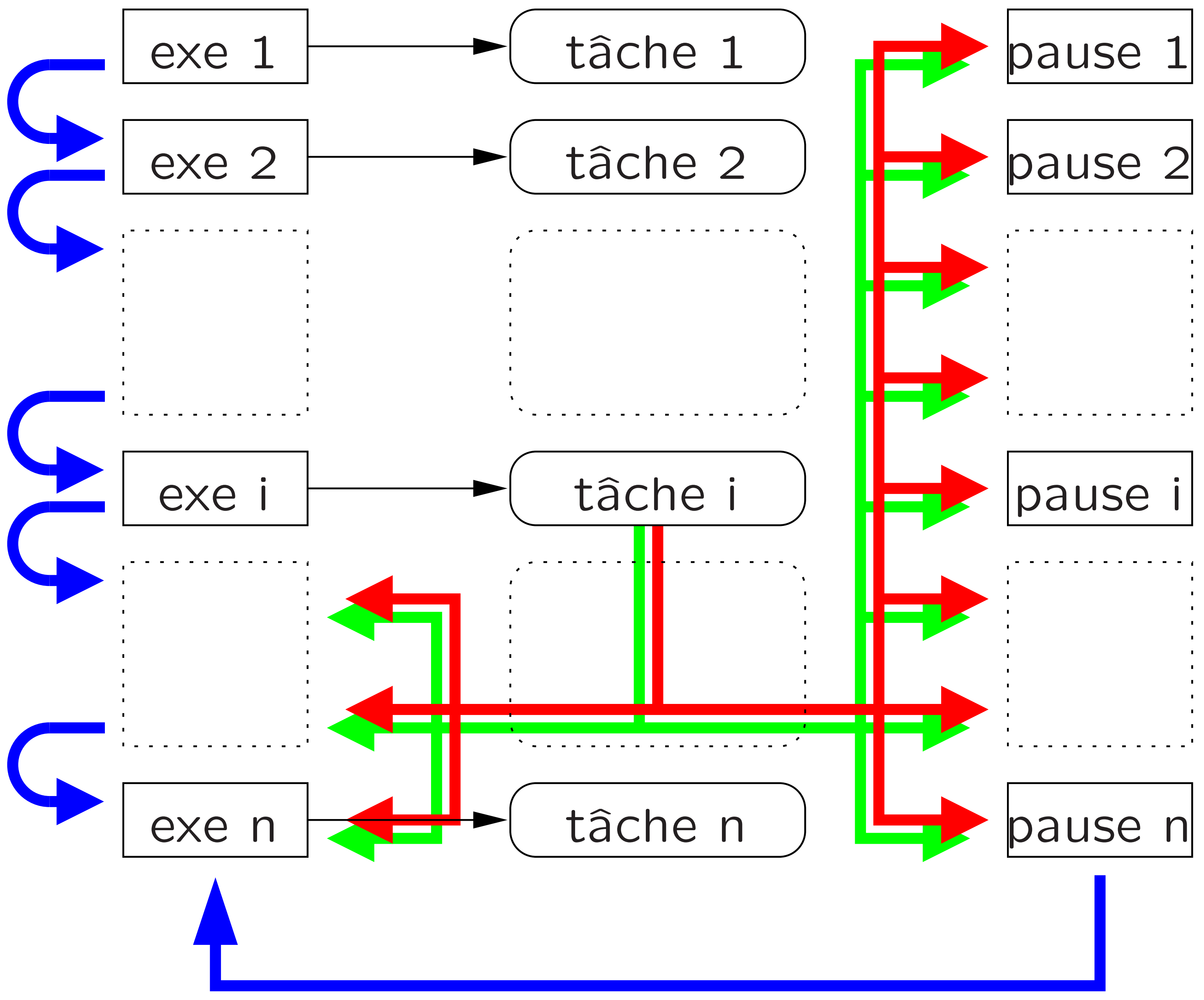
Le code OC est efficace mais très gros (explosion combinatoire)

Le code SC est petit mais très lent

Nouvelle méthode de compilation hybride :

- Identification des points d'arrêt du programme
- Ordonnancement de ces points d'arrêt
- Génération de code C efficace et petit

Modèle d'exécution



Tout n'est pas si simple :

- Programmes qui n'ont pas de comportement :

  ```
  present O else emit O end
  ```

  ➥ programmes non réactifs

- Programmes qui ont plusieurs comportements :

  ```
  present O then emit O end
  ```

  ➥ programmes non déterministes

Le compilateur vérifie qu'il n'y a pas de dépendance instantanée entre des signaux

➥ Simulation tri-valuée des signaux : présent, absent ou inconnu

Validation de l'hypothèse de synchronisme

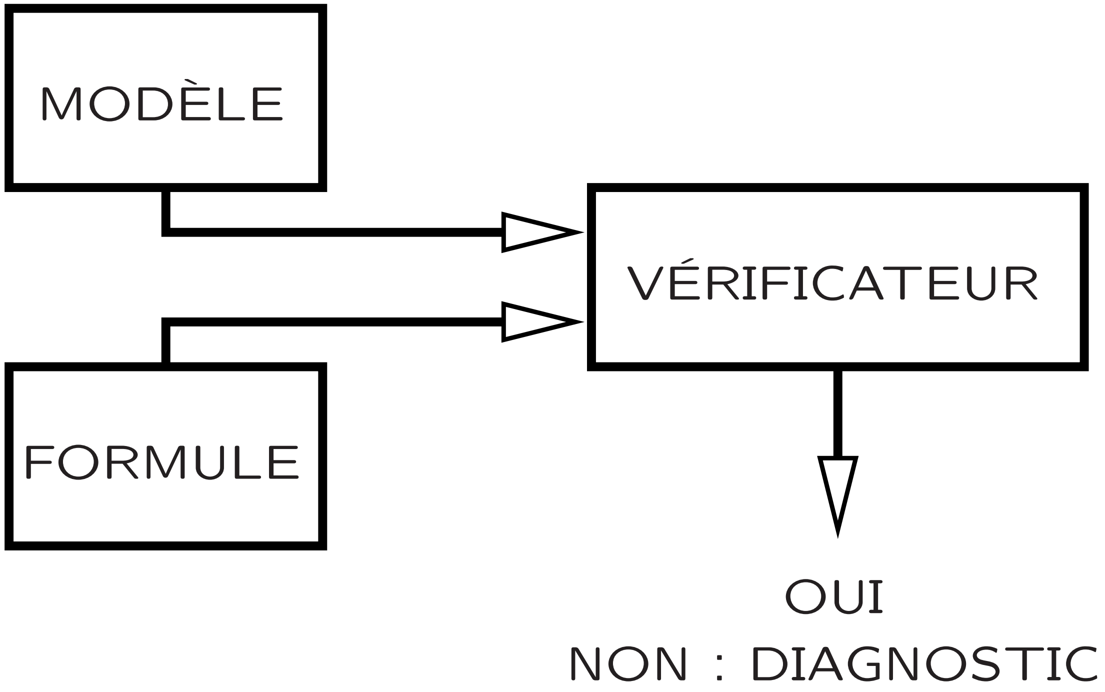
Connaissance des fréquences maximales des entrées et des contraintes temporelles

L'automate fini (explicite ou implicite) permet de connaître exactement le temps de réaction du programme

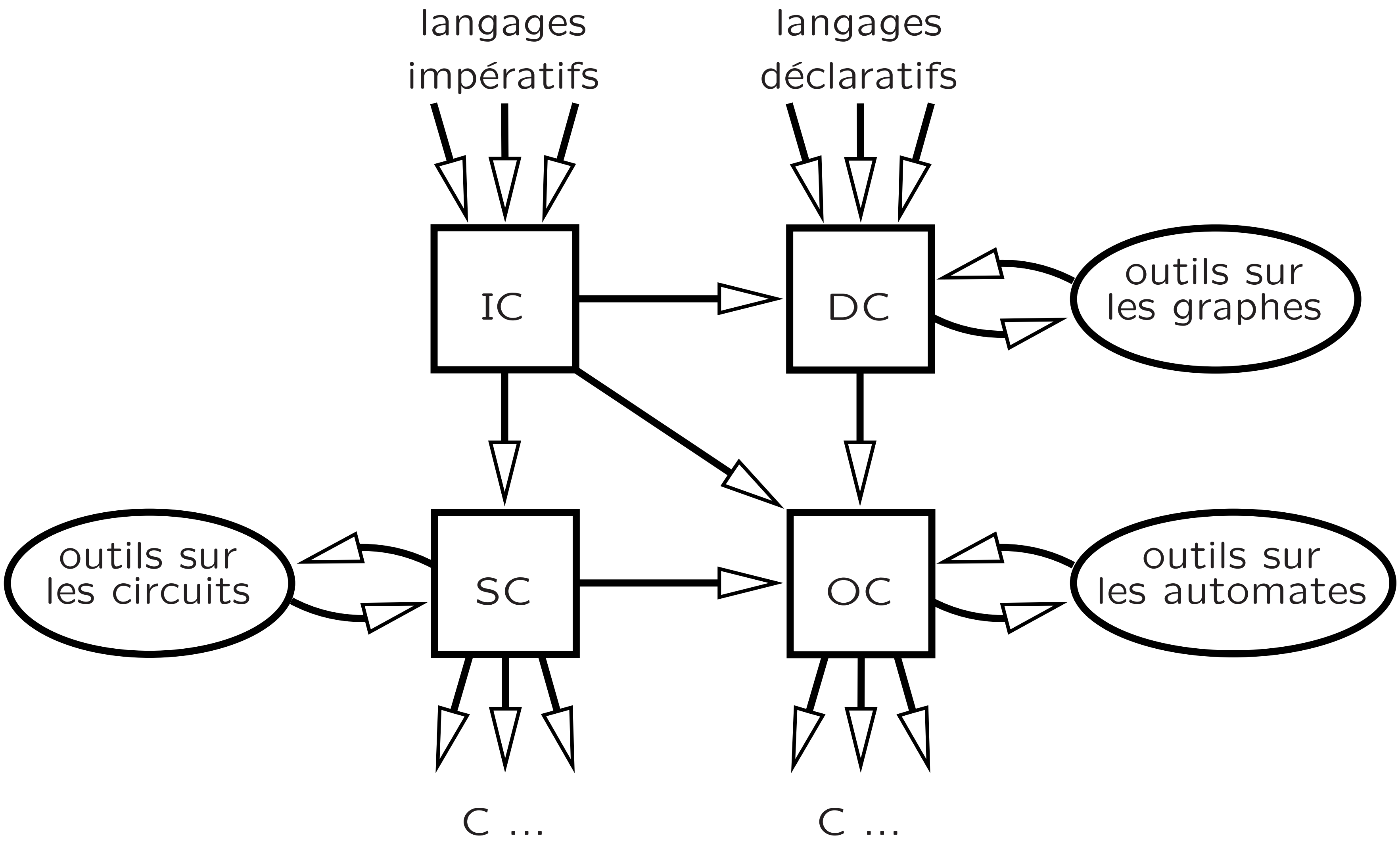
➥ A comparer avec les contraintes temporelles

Vérification basée sur le modèle (model-checking)

Grâce à l'automate fini (explicite ou implicite) (encore une fois)



Formats communs



Références

- Esterel Technologies, Nice

  Gérard Berry, Xavier Fornari et Amar Bouali

  `www.esterel-technologies.com`

  `www.esterel.org`

- « The Foundations of Esterel »

- « The Esterel V5 Language Primer »

- « The Constructive Semantics of Pure Esterel »