UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Sophia Antipolis
B.P. 109
06561 Valbonne Cedex
France
Tél.: 93 65 77 77

Rapports de Recherche

N°1588

Programme 2
Calcul symbolique, Programmation
et Génie logiciel

RÉSEAUX DE PROCESSUS RÉACTIFS

Frédéric BOUSSINOT

Janvier 1992

Réseaux de Processus Réactifs

Nets of Reactive Processes

Frédéric Boussinot
Ecole Nationale Supérieure des Mines de Paris
Centre de Mathématiques Appliquées
Sophia-Antipolis
06565 Valbonne, France
email: fb@cma.cma.fr

Résumé. On introduit le modèle des réseaux de processus réactifs, on décrit une implémentation de ces réseaux et on leur donne une sémantique mathématique dans le cadre dénotationnel. On étudie l'implémentation des langages synchrones Lustre et Signal en considérant leurs programmes comme des exemples de réseaux réactifs.

Abstract. We introduce the model of nets made of reactive processes. We describe an implementation of these nets and give them a denotational semantics. We study the implementation of the synchronous languages Lustre and Signal, considering their programs as reactive nets.

1 Introduction

Réseaux de processus séquentiels.

Les réseaux de processus séquentiels, introduit par G. Kahn dans [8], sont essentiellement *asynchrones* : ils sont formés de programmes appelés *processus*, qui s'exécutent indépendament les uns des autres et qui communiquent en plaçant des messages dans des files "premier entré, premier sorti" (FIFO). Ces files appelées *canaux*, sont de *taille non bornée*. Un processus a une mémoire propre et lors de son exécution, il peut mettre un message dans un de ses canaux de sortie (toujours possible puisque les canaux sont de taille non bornée) ou bien tenter de prendre un message dans un de ses canaux d'entrée. Les processus sont *séquentiels et déterministes* : il ne peuvent tenter de prendre des messages sur plusieurs canaux en concurrence et un processus qui tente de prendre un message dans un canal vide *se bloque* jusqu'à ce que le canal ne soit plus vide. Un réseau est formé en interconnectant des processus par des canaux de telle façon qu'il y ait au plus un processus pouvant mettre des messages dans chaque canal (le *producteur* du canal) et au plus un processus pouvant prendre des messages dans un canal (le *consommateur* du canal). Les canaux sans producteur sont les entrées du réseau et les canaux sans consommateur, ses sorties. La structure des réseaux c'est à dire le nombre de processus et de canaux, peut *évoluer dynamiquement* (un tel réseau calculant la suite des nombres premiers est décrit dans la section 3). Le *déterminisme* est la propriété fondamentale des réseaux de processus séquentiels : la suite des valeurs qui passent dans un canal donné, appelée *histoire* du canal, est fonctions des histoires des canaux d'entrée du réseau. Une sémantique mathématique fonctionnelle ("dénotationnelle") des réseaux de processus séquentiels est décrite dans [8].

Il existe plusieurs stratégies d'exécution des réseaux séquentiels, dont l'exécution parallèle dans laquelle les processus s'exécutent simultanément. Une exécution "par nécessité" est également possible; elle consiste à exécuter le producteur d'un canal parceque le consommateur a besoin d'une information provenant de ce canal et que celui-ci est vide. Le producteur est alors exécuté jusqu'à la production d'une valeur dans le canal considéré. Dans tous les cas, le comportement des réseaux séquentiels est indépendant de la stratégie d'exécution choisie (propriété de déterminisme).

Seules deux actions sont autorisées sur un canal : mettre une information dans un canal, ou prendre une information d'un canal non vide. Le point important est qu'un processus qui tente de prendre une information dans un canal vide *reste bloqué* jusqu'à ce qu'une information soit mise dans le canal. En particulier, les processus n'ont pas le droit de *tester si un canal est vide*. Permettre ce test introduit le nondéterminisme : le résultat dépend de la rapidité d'exécution des processus. Si le producteur d'un canal est plus "lent" que son consommateur, le canal peut être vu comme vide par le consommateur, alors que dans le cas contraire, il aurait pu être rempli. En absence de test du canal vide, il n'est pas possible de traiter deux canaux en leur donnant des priorités différentes (exemple : un serveur d'impression, avec une file de fichiers prioritaires et une file de fichiers non prioritaires). Ceci limite fortement la puissance d'expression des réseaux séquentiels et par conséquent, leur utilisation.

Réseaux de processus réactifs.

Un programme réactif est un programme qui contient des *points d'arrêt* : à chaque activation, l'exécution part du point d'arrêt courant et s'arrête sur un nouveau point d'arrêt qui sera le point de départ lors de la prochaine activation. Dans ce texte, on considère un nouveau modèle de réseaux formés de processus qui sont des programmes réactifs séquentiels et déterministes. La notion de réaction des processus permet de définir une notion d'*instant* global d'un réseau : un instant consiste à faire réagir tous les processus qui le composent. Les réseaux de processus réactifs sont exécutés de manière "synchrone" : tous les processus sont activés et l'instant courant est terminé lorsqu'ils ont tous fini de réagir. Le réseau est alors prêt pour l'exécution à l'instant suivant. Il est à noter qu'un processus peut prendre ou mettre plusieurs valeurs dans un canal au cours d'un même instant.

La notion d'instant permet d'introduire le "test du canal vide" dans le modèle des réseaux réactifs, sans perdre la propriété de déterminisme. En effet on peut donner une portée temporelle au test : un canal est vide pendant un instant, si il était vide au début de l'instant et si aucune valeur n'y a été mise pendant l'instant. La possibilité de définir un tel test accroit la puissance d'expression par rapport aux réseaux séquentiels et permet en particulier de traiter des canaux avec des priorités différentes.

Le langage "Reactive C" (RC [3]) est une extension réactive du langage C qui permet l'écriture de programmes réactifs. RC permet d'implémenter des systèmes fondés sur des mécanismes spécifiques d'activation et de réaction. En particulier RC peut être utilisé pour implémenter des exécutions dans lesquelles tous les composants doivent réagir à chaque instant (exécutions " synchrones"). Dans ce texte, on utilise RC pour implémenter les réseaux de processus réactifs. Chaque processus est

implémenté comme une procédure réactive de RC. Un réseau est également une procédure réactive qui mélange (en utilisant la primitive `merge` de RC) les comportements des processus qui le composent.

Les langages Lustre et Signal.

RC, est comme C un langage généraliste de bas niveau. Au contraire, les langages *synchrones* Esterel[1], Lustre[6] et Signal[9] sont des langages spécialisés pour le "temps réel". Lustre et Signal sont des langages "flot de données" dans lesquels les programmes sont des *systèmes d'équations récursives* dont toutes les variables doivent être évaluées à chaque instant. Par exemple, le système Lustre suivant calcule dans la variable `x` la suite de fibonnacci :

```
x1 = 0 -> pre x;
x2 = pre x1 + pre x;
x = 1 -> x2;
```

La sémantique des langages synchrones "flot de données" a été liée au modèle des réseaux séquentiels, par exemple dans [5]. On va montrer que les programmes Lustre et Signal sont des exemples de réseaux de processus réactifs. Dans le cas particulier de ces deux langages, on ne peut pas mettre ou prendre plus d'une valeur dans un canal lors d'un instant.

Organisation du texte.

Le papier est organisé comme suit : dans la section 2 on introduit les réseaux de processus réactifs ainsi que le test du canal vide dans ces réseaux. La section 3 contient une implémentation des réseaux de processus réactifs réalisée en RC. Trois exemples illustrent cette implémentation. Dans la section 4 on décrit une sémantique dénotationnelle des réseaux réactifs. Dans la section 5 on montre comment implémenter les langages Lustre et Signal dans le cadre des réseaux réactifs, à l'aide d'une information spéciale qui code l'absence de valeur dans un canal à un instant. Enfin, dans la section 6 on étudie l'implémentation de Lustre et Signal sans utiliser d'information spéciale et on montre une équivalence : ces deux langages permettent d'exprimer le "test instantané du canal vide" dont la puissance entraine l'existence de réseaux incohérents.

2 Réseaux réactifs

Dans cette section, on considère des réseaux dont les processus sont des programmes réactifs. Le processus `Pre` qui recopie son entrée sur sa sortie avec un décalage d'un instant, est un exemple typique de processus réactifs (il correspond à la primitive de même nom en Lustre). A sa première activation, `Pre` prend une valeur sur son entrée et ne produit rien. Aux activations suivantes, il met sur sa sortie la valeur prise précedemment et prend une nouvelle valeur pour l'activation suivante. Indépendemment des instants, `Pre` est une simple identité : la suite des valeurs sur son entrée est la même que celle sur sa sortie. Le processus `Pre` ne prend donc son sens que par rapport à la notion d'instant.

L'exécution d'un processus réactif s'arrête dans un des trois cas suivants :

- Lorsque le processus termine.
- Sur un point d'arrêt explicite.
- Sur une instruction de prise dans un canal, si celui-ci est vide.

Un réseau réactif consiste en la mise en parallèle de processus réactifs. Exécuter dans l'instant un réseau réactif signifie activer chacun des composants du réseau. Le réseau peut alors être à nouveau exécuté à l'instant suivant. Un instant global du réseau correspond donc à un instant local de chacun des composants.

Les processus séquentiels sont des cas particuliers de processus réactifs, sans point d'arrêt explicite : seules les lectures bloquantes dans des canaux vides séparent les instants. Un réseau de processus séquentiels considéré du point de vue réactif a toutefois une interprétation différente de celle originelle de Kahn : dans un réseau réactif, les instants sont communs à tous les processus. En particulier, il ne peut y avoir de situation de "dérive" dans laquelle un processus prend de l'avance sur les autres en exécutant plusieurs de ses instants locaux alors que d'autres n'en exécutent aucun.

La figure 1 montre un réseau de processus réactifs qui utilise le processus `Pre` et calcule la suite de fibonnacci sur l'entrée du processus de visualisation `Out`. Les canaux sont représentés par des flèches reliant le producteur d'information au consommateur.

Dans ce réseau :

- `0` désigne le processus qui envoie un 0 sur sa sortie à chaque instant.
- `1` désigne le processus qui envoie un 1 sur sa sortie à chaque instant.
- `+` désigne le processus qui à chaque instant, prend une valeur successivement sur son entrée gauche puis droite, puis met la somme sur sa sortie.
- Le processus `Fol` ("follow") laisse passer une unique valeur provenant de son entrée gauche,

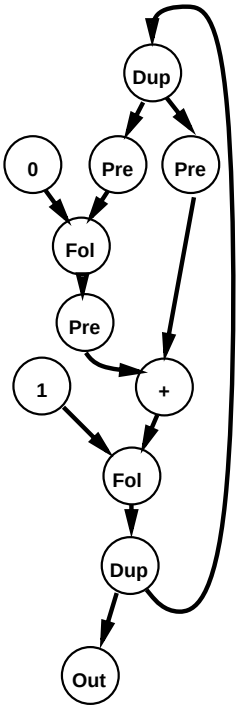

Figure 1: Réseau de processus réactifs

puis aux instants suivants, il recopie son entrée droite sur sa sortie.

- `Dup` duplique son entrée sur ses deux sorties.
- `Out` imprime son entrée.

On va maintenant introduire le "test du canal vide" dans les réseaux réactifs.

Test du canal vide.

Le déterminisme des réseaux de processus séquentiels et la possibilité de leur associer simplement une sémantique dépendent fortement des conditions suivantes :

- Impossibilité de tester si un canal est vide.
- Impossibilité d'attendre sur plusieurs canaux simultanément.
- Chaque canal a au plus un producteur et au plus un consommateur.
- Les canaux sont de taille non bornée.

En l'absence d'une de ces conditions, le comportement des réseaux séquentiels peut dépendre des vitesses relatives d'exécution des processus et devenir nondéterministe [8].

Dans les réseaux réactifs, on peut relaxer la première condition tout en préservant le déterminisme. En effet un canal est considéré vide non plus comme dans les réseaux séquentiels, parce qu'une étape d'exécution le détecte ainsi, mais parce qu'il reste vide durant l'instant courant.

On peut envisager deux tests différents du canal vide :

- Dans le premier test, appelé "*différé*", la réaction dans le cas d'un canal vide, n'a lieu *qu'à l'instant suivant*; par contre lorsqu'une valeur est présente, l'exécution de l'instant courant se poursuit normalement. L'idée du test différé est la suivante : on ne peut savoir qu'un canal est vide que lorsqu'on est sûr que son producteur ne le remplira pas. C'est le cas à la fin de l'instant qui est commune à tous les processus. En conséquence, on ne peut traiter le cas où le canal est vide qu'à la prochaine activation.
- Le second test, appelé "*instantané*" correspond à ne pas attendre l'instant suivant pour réagir, dans le cas d'un canal vide. Il pose des problèmes de cohérence au cas où la réaction consiste justement à remplir le canal. La présence de ce test introduit donc des réseaux *incohérents* tel celui de la figure 2. Dans ce réseau, le processus `TestInst` utilise le test instantané sur son entrée et met la valeur booléenne "vraie" (notée "**tt**") sur sa sortie si l'entrée n'est pas vide, et la valeur booléenne "faux" (notée "**ff**") sinon. Le processus `P0` met 0 sur sa sortie lorsqu'il recoit **ff** et ne met rien sur sa sortie lorsqu'il recoit **tt**. Au premier instant, si l'entrée de `TestInst` est non vide, c'est que `P0` a mis un 0 et donc qu'il a reçu **ff** ce qui est contradictoire avec le fait que l'entrée de `TestInst` est non vide. Inversement, si l'entrée de `TestInst` est vide, c'est que `P0` a reçu **tt** ce qui est contradictoire avec le fait que l'entrée de `TestInst` est vide.

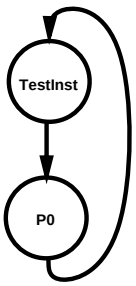

Figure 2: Réseau incohérent

Les réseaux incohérents doivent être éliminés : soit il existe un moyen de déterminer statiquement

si un réseau est ou non cohérent avant de l'exécuter, soit on est obligé de rejeter les réseaux incohérents lors de leur exécution.

Face à ce problème de cohérence, on a choisi de ne considérer que le test différé du canal vide. Ainsi les processus réactifs peuvent tester l'état de leurs canaux d'entrée en utilisant l'instruction `Delayed` définie de la manière suivante :

- Dans le cas où le canal n'est pas vide, `Delayed` retourne **tt** et termine dans l'instant,
- dans le cas contraire, `Delayed` retourne **ff** mais bloque l'exécution du processus qui l'exécute jusqu'à l'instant suivant.

Le test de plusieurs canaux peut également être effectué en introduisant de nouvelles primitives. Par exemple, la primitive `Delayed2` teste deux canaux :

- Dans le cas où les deux canaux ne sont pas vides, `Delayed2` retourne deux valeurs **tt** et termine,
- dans le cas contraire, `Delayed2` retourne **ff** ou **tt** pour chaque canal suivant qu'il est vide ou non, mais bloque l'exécution jusqu'à l'instant suivant.

Un exemple d'utilisation de `Delayed2` est décrit dans la section 6.

Situation du problème en Esterel.

Les réseaux incohérents correspondent aux programmes Esterel incohérents, par exemple :

```
signal S in
   present S else emit S end
end
```

Dans les deux cas, on teste une absence et on réagit au cours du même instant à cette absence en la contredisant.

Le retard de réaction à l'instant suivant, dans le cas d'un canal vide, permet d'éviter les problèmes d'incohérence. Une approche similaire est possible en Esterel. Elle consiste à n'avoir que des instructions `present` dans lesquelles la branche `else` est systématiquement retardée d'un instant. Le retard d'un instant est implémenté par l'attente du signal d'activation `tick`. Les instructions de test de présence d'un signal ont donc la forme générale suivante :

```
present S then ...
else await tick; ...
end
```

Cette restriction d'utilisation des tests de signaux correspond en fait à ne permettre que le "watching faible", qui exécute son corps dans tous les cas, et à rejetter le "watching fort" d'Esterel. Avec cette restriction, les problèmes de causalité sont éliminés.

On retrouve également une approche semblable dans [7] où la réaction est fondée sur un mécanisme de pulsation dans lequel le traitement de l'absence d'un événement est reportée à l'instant suivant.

3 Implémentation en RC des réseaux réactifs

Les processus réactifs peuvent être simplement implémentés en RC en codant les points d'arrêt explicites des processus par des instructions `stop`. On illustre cette implémentation sur trois exemples : le programme de calcul de la suite de fibonnacci décrit précedemment, un programme de calcul de la suite des nombres premiers qui est un exemple de réseau évoluant dynamiquement, et une utilisation du test différé du canal vide.

Suite de Fibonnacci.

On reprend le réseau de la figure 1. On code chacun des processus réactifs par une *procédure réactive* qui a en paramètre les canaux d'entrée et de sortie du processus. L'exécution de la procédure réactive `Get` permet de prendre une information dans un canal. Cette procédure termine quand une valeur peut être prise et elle bloque l'exécution sinon. Dans ce dernier cas, elle est à nouveau exécutée à l'instant suivant. Par contre, la mise d'une information dans un canal, notée `Put`, est codé par une fonction C puisque les canaux étant de taille non bornée, cette action est toujours immédiatement possible.

Le processus `Pre` est codé en RC par :

```
rproc Pre(in,out)
Channel in, out;
{
   rauto int val;
   for(;;){
      exec Get(in,&val);
      stop;
      Put(out,val);
   }
}
```

Une valeur est prise sur le canal d'entrée `in` et elle est mise à l'instant suivant sur le canal de sortie `out` (à noter que la variable `val` doit être déclarée comme `rauto` pour maintenir sa valeur entre l'instant où elle est prise et l'instant suivant où elle est mise dans le canal de sortie).

A chaque instant le processus `Const` met une constante sur son canal de sortie. Le processus `0` est une instance de ce processus avec 0 en paramètre et `1` est une instance avec 1 en paramètre. Le processus `Const` est codé en RC par :

```
rproc Const(value,out)
int value;
Channel out;
{
   for(;;){
      Put(out,value);
      stop;
   }
}
```

Le processus `Fol` laisse passer une unique valeur provenant de son entrée gauche, puis recopie son entrée droite sur sa sortie. Le code RC est :

```
rproc Fol(leftIn,rightIn,out)
Channel leftIn, rightIn, out;
{
   int val;
   exec Get(leftIn,&val);
   Put(out,val);
   for(;;){
      stop;
      exec Get(rightIn,&val);
      Put(out,val);
   }
}
```

Le réseau est obtenu en exécutant en parallèle les processus interconnectés par des canaux :

```
rproc FIB(){
 Channel C[13];
 int i;
 for(i=0;i<13;i++) InitChannel(C[i]);

 merge exec Plus(C[3],C[9],C[10]);
 merge exec Const(1,C[1]);
 merge exec Const(0,C[4]);
 merge exec Dup(C[5],C[2],C[6]);
 merge exec Pre(C[6],C[7]);
 merge exec Fol(C[4],C[7],C[8]);
 merge exec Pre(C[8],C[9]);
 merge exec Fol(C[1],C[10],C[11]);
 merge exec Dup(C[11],C[12],C[5]);
 merge exec Pre(C[2],C[3]);
       exec Out(C[12]);
}
```

L'opérateur `merge` de RC est utilisé pour construire le réseau. On modélise ainsi la mise en parallèle des composants qui ne communiquent que par les canaux. Il est à noter que dans l'implémentation réalisée, l'ordre des instructions `merge` qui composent le corps de `FIB` n'est pas significatif.

L'exécution de `FIB` donne, comme on s'y attend :

```
1
1
2
3
5
8
13
21
.....
```

Calcul de nombres premiers.

Un programme de calcul des nombres premiers par la méthode du crible d'Eratosthène est décrit dans [8]. Trois processus sont mis en parallèle :

- Le processus `INTEGERS` produit la suite des nombres entiers à partir de 2.
- Le processus `Out` qui visualise la suite des nombres produits.
- Le processus `SIFT` recoit un nombre premier sur son entrée. Il crée un "filtre" associé à ce nombre et s'exécute en séquence avec ce filtre. Les filtres sont tous des instances du même processus `FILTER` qui élimine de son entrée tous les multiples d'un nombre premier passé en paramètre. Le processus `SIFT` est défini récursivement et la structure du réseau évolue dynamiquement. La reconfiguration de `SIFT` est illustrée sur la figure 3.

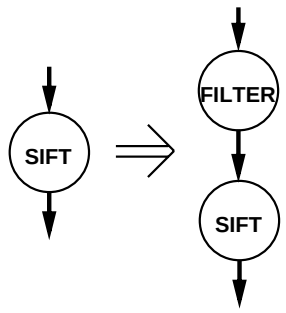

Figure 3: Reconfiguration de `SIFT`

Le programme de calcul des nombres premiers exprimé dans le langage introduit dans [8], est simple et concis. Voici ce code :

```
Process INTEGERS out QO;
   Vars N; 1 -> N;
   repeat
     INCREMENT N; PUT(N,QO)
   forever
Endprocess;
```

```
Process FILTER PRIME in QI out QO;
   Vars N;
   repeat
      GET(QI) -> N;
      if (N MOD PRIME) != 0)
      then PUT(N,QO) close
   forever
Endprocess;

Process SIFT in QI out QO;
   Vars PRIME; GET(QI) -> PRIME;
   PUT(PRIME,QO);
   doco channels Q;
      FILTER(PRIME,QI,QO);
      SIFT(Q,QO);
   closeco
Endprocess;

Process OUTPUT in QI;
   repeat
      PRINT(GET(QI))
   forever
Endprocess;

Start
  doco channels Q1, Q2;
    INTEGERS(Q1);
    SIFT(Q1,Q2);
    OUTPUT(Q2);
  closeco;
```

Le code RC.

On traduit en RC chacun des processus. Le code RC du processus `FILTER` est :

```
rproc FILTER(prime,in,out)
int prime;
Channel in, out;
{
  int val;
  for(;;){
    exec Get(in,&val);
    if(val%prime!=0) Put(out,val);
  }
}
```

Le processus `SIFT` est défini récursivement par :

```
rproc SIFT(in,out)
Channel in, out;
{
  rauto Channel intern;
  rauto int prime;
  exec Get(in,&prime);
  Put(out,prime);
  InitChannel(intern);
  merge
    exec FILTER(prime,in,intern);
    exec SIFT(intern,out);
}
```

Le code du processus `INTEGERS` est le suivant :

```
rproc INTEGERS(out)
Channel out;
{
    rauto int val;
    val = 1;
    for(;;){
       Put(out,++val);
       stop;
    }
}
```

Le réseau est formé en mettant en parallèle le producteur de nombres, le processus `SIFT` et le processus de visualisation. Son code est le suivant :

```
rproc SIEVE(){
   rauto Channel intern1, intern2;
   InitChannel(intern1);
   InitChannel(intern2);

   merge exec SIFT(intern1,intern2);
   merge exec INTEGERS(intern1);
         exec Out(intern2);
}
```

L'exécution de `SIEVE` donne :

```
2
3
5
7
11
13
17
19
.....
```

On remarque que la traduction en RC du langage de [8] est immédiate, en particulier en ce qui concerne la récursivité.

Test du canal vide.

Dans ce paragraphe on considère un exemple de réseau réactif utilisant le test différé du canal vide. On reboucle la sortie sur l'entrée d'un processus `NOT` qui teste si son entrée est vide et met "0" sur sa sortie si c'est le cas. Ainsi il produit une suite de "0" séparés par deux instants. Le réseau réactif `FLIPFLOP` est le suivant :

```
rproc NOT(in,out)
Channel in, out;
```

```
{
   rauto int b;
   for(;;){
     exec Delayed(in,&b);
     if (b==FALSE) Put(out,0);
     else{
         int v;
         exec Get(in,&v);
     }
     stop;
   }
}

rproc FLIPFLOP()
{
    rauto Channel i1,i2,i3;
    InitChannel(i1);
    InitChannel(i2);
    InitChannel(i3);

    merge exec NOT(i1,i2);
    merge exec Dup(i2,i1,i3);
          exec Out(i3);
}
```

La sortie produite en fonction des instants est la suivante :

```
instant 1:
instant 2: 0
instant 3:
instant 4:
instant 5: 0
instant 6:
instant 7:
instant 8: 0
instant 9:
...
```

Au premier instant, les canaux sont vides et rien n'est produit. Au second instant, `NOT` produit une valeur puisque son canal d'entrée était vide à l'instant précedent. Au troisième instant, la valeur produite au second instant est consommée et rien n'est produit. Le quatrième instant est semblable au premier.

Conclusion.

On a décrit les grandes lignes d'une implémentation des réseaux de processus réactifs. Les réseaux dont la structure évolue dynamiquement ainsi que le test différé du canal vide sont implémentés.

L'implémentation décrite peut être également vue comme une technique particulière (synchrone) d'exécution des réseaux de processus séquentiels. Elle illustre la possibilité d'utiliser le langage réactif RC pour implémenter des modèles avec communication asynchrone.

4 Sémantique des réseaux réactifs

Dans cette section on donne une sémantique aux réseaux réactifs. Une sémantique dénotationnelle des réseaux de processus séquentiels a été donnée dans [8]. Dans cette sémantique, les valeurs des canaux sont des suites finies ou infinies (dénombrables), appelées *histoires* et ordonnées par ordre préfixe ("abc" plus petit que "abcd" et incomparable avec "abd"). La sémantique d'un processus séquentiel est construite à partir de *fonctions continues*, c'est-à-dire croissantes et préservant les limites (voir [8] pour plus de détails). Les réseaux sont traduits en des systèmes d'équations définies récursivement. L'histoire associée à chaque variable est le *plus petit point fixe* de l'équation qui la définit. Cette approche est particulièrement simple et élégante dans la mesure où elle relie les notions de réseau et de processus à la notion mathématique de fonction.

La sémantique de [8] n'est pas adaptée aux réseaux réactifs : l'histoire d'un canal est simplement la suite des valeurs qui transitent à travers ce canal, indépendemment des instants où elles sont produites. Le processus `pre` serait ainsi une simple identité, ce qui ne correspond manifestement pas à sa sémantique intuitive de décalage d'un instant.

On va maintenant décrire une solution qui correspond à associer aux informations produites leur instant de production.

Le domaine intuitif.

Intuitivement, le domaine d'interprétation $\mathcal{T}$ est celui des suites de valeurs avec leur instant de production. Pour simplifier, on supposera dans la suite que les valeurs sont toujours entières. On note l'instant de production d'une valeur comme indice de cette valeur. Par exemple, la suite $1_1 2_2 3_3 ...$ signifie que la valeur i est produite à l'instant numéro i. Plusieurs valeurs peuvent être produites au cours du même instant comme dans $1_1 2_1 3_3 ...$ où 1 et 2 sont produites pendant l'instant 1. A noter que dans cette suite, aucune valeur n'est produite à l'instant 2. Dans ce cadre, la fonction suivante correspond à `Pre` :

$$pre(x_{i_0} y_{i_1} z_{i_2} ...) = x_{(1+i_0)} y_{(1+i_1)} z_{(1+i_2)} \cdots$$

Ainsi, on a par exemple :

$$pre(1_0 2_0 3_1 4_1 5_2 6_2 ...) = 1_1 2_1 3_2 4_2 5_3 6_3 ...$$

Techniquement, on montre (cf. [2]) que le domaine $\mathcal{T}$ est isomorphe au domaine $\mathcal{D} = (\mathcal{N} \cup \tau)^\omega$ des suites finies ou infinies dénombrables dont les éléments sont :

- Des valeurs entières, éléments de $\mathcal{N}$.
- Le symbole spécial τ qui représente la frontière entre deux instants.

Le plongement des éléments de $\mathcal{T}$ dans $\mathcal{H}$ est immédiat. Par exemple, $1_1 2_2 3_3 ...$ correspond à la suite $\tau 1 \tau 2 \tau 3 \tau ...$ et $1_1 2_1 3_3 ...$ à $\tau 12 \tau 3 \tau ....$ A noter que la suite vide de $\mathcal{T}$ correspond à la suite infinie τ^ω de $\mathcal{H}$ et que les éléments maximaux de $\mathcal{T}$ sont les suites infinies de $\mathcal{H}$.

L'introduction de τ permet de distinguer les processus qui bloquent le passage des instants de ceux qui ne le font pas. Considérons le processus suivant qui "boucle instantanément" (il "diverge") :

```
rproc InstLoop(out)
Channel out;
{
   lab: goto lab;
}
```

La sémantique de ce processus est la suite vide ϵ : il ne permet pas le passage des instants. La présence de `InstLoop` dans un réseau a pour effet de bloquer l'exécution de tous les processus du réseau au premier instant.

Comme le processus précédent, le processus `Halt` ne produit rien; cependant, il ne bloque pas le passage des instants. Il est défini par :

```
rproc Halt(out)
Channel out;
{
   for(;;) stop;
}
```

La sémantique de ce processus est la constante τ^ω.

Un autre exemple de processus qui "diverge" en bloquant le passage des instants est :

```
rproc Diverge(out)
Channel out;
{
   Put(out,1);
   exec Diverge(out);
}
```

La sémantique de ce processus défini récursivement est la suite vide ϵ.

L'introduction de τ est nécessaire pour distinguer les processus qui divergent de ceux qui ne bloquent pas le passage du temps. Cependant elle n'est pas suffisante car elle donne une sémantique de divergence à des réseaux qui s'exécutent sans problème. Considérons le processus identité suivant :

```
rproc Ident(in,out)
Channel in, out;
{
   int v;
   for(;;){
     exec Get(in,&v);
     Put(out,v);
   }
}
```

On devrait avoir :

$$Ident(1) = 1$$

et

$$Ident(\tau) = \tau$$

Puisque $Ident$ doit être continue, on devrait avoir nécessairement :

$$Ident(\epsilon) = \epsilon$$

Mais alors la solution du réseau obtenu en rebouclant la sortie de `Ident` sur son entrée, c'est à dire le plus petit point fixe de $X = Ident(X)$, serait ϵ. Ainsi, on identifierait ce réseau avec `InstLoop`, ce qui n'est pas cohérent avec la sémantique opérationnelle des réseaux réactifs (la présence d'un processus identité rebouclé sur lui-même ne doit pas bloquer le temps et donc interdire aux autres processus de fonctionner).

La solution proposée consiste à ajouter un symbole spécial $\star$ qui est la trace de la détection d'un canal vide. La présence de ce symbole permet de savoir qu'un processus ne diverge pas. Le domaine $\mathcal{H}$ est donc défini par :

$$\mathcal{H} = (\mathcal{D} \cup \mathcal{D}\star, \prec, \epsilon)$$

où :

- Les éléments de $\mathcal{D}\star$ sont les suites finies de $\mathcal{D}$ auxquelles on a rajouté le symbole $\star$ à la fin.
- ϵ est la suite vide.
- L'ordre $\prec$ est défini par : $X \prec Y$ si X est préfixe de Y ou bien si Y est obtenu à partir de X en remplacant $\star$ par une suite non vide.

En reprenant l'exemple précédent, on a maintenant :

$$Ident(\epsilon) = \star$$

On a gardé trace du fait que le processus ne diverge pas mais est bloqué sur son canal d'entrée sur lequel on n'a aucune information. Ainsi le calcul de la sémantique du réseau ne restera pas indéfiniment bloqué sur ϵ.

Une histoire qui se termine par $\star$ signifie une absence de divergence. Le symbole $\star$ est intermédiaire entre la divergence ϵ et la présence ou l'absence d'une valeur : on a $\star \prec n$ et $\star \prec \tau$.

Sémantique des processus.

A chaque processus est associée une fonction continue qui est sa sémantique. Les sémantiques des processus sont typées par :

$$\mathcal{H}^n \rightarrow \mathcal{H}^p$$

où n est le nombre d'entrées du processus et p le nombre de ses sorties. Par exemple, la sémantique de `Ident` est la fonction définie par : si X est une suite finie d'entiers ou de τ, alors $Ident(X) = X\star$, autrement $Ident(X) = X$. Cette fonction est bien continue.

Comme autre exemple, considérons le processus suivant qui vide son entrée :

```
rproc Clear(in,out)
Channel in,out;
{
   int v;
   for(;;) exec Get(in,&v);
}
```

La sémantique de ce processus est la fonction $\lambda X.(\tau^{|X|})\star$, où $|X|$ est le nombre de τ dans X.

Les fonctions définie récursivement à partir de fonction continues sont également continues. Par exemple, la fonction *current* (qui correspond à l'opérateur de même nom en Lustre, voir plus loin) est définie récursivement en utilisant le préfixage d'une suite, noté "$\bullet$", par :

$$current(X) \quad = \quad current_\tau(X)$$

où :

$$\begin{aligned} current_a(\epsilon) &= \epsilon \\ current_a(\star) &= \star \\ current_a(n \bullet X) &= n \bullet current_n(X) \\ current_a(\tau \bullet X) &= \tau \bullet cur_a(X) \end{aligned}$$

et la fonction auxiliaire *cur* est définie par :

$$\begin{aligned} cur_a(\epsilon) &= \epsilon \\ cur_a(\star) &= \star \\ cur_a(n \bullet X) &= n \bullet current_n(X) \\ cur_a(\tau \bullet X) &= a\tau \bullet current_a(X) \end{aligned}$$

La fonction *current* est donc bien continue.

Sémantique des réseaux.

La sémantique d'un réseau est définie de la même façon que dans [8], à partir des sémantiques de ses composants. Cependant on ne considère ici que les réseaux clos, sans entrée ni sortie. La construction de la sémantique du réseau est plus complexe que dans [8] car elle modélise une exécution qui enchaine deux phases : la première consiste à exécuter tous les processus jusqu'à ce que tous aient soit terminés leur instant, soit soient bloqués sur un accés (par `Get` ou `Delayed`) à un canal vide. La seconde phase consiste alors à forcer les processus bloqués sur des canaux vides à terminer leur instant. La fonction $\top$ modélise la seconde phase : elle signifie "terminer l'instant" dans le cas d'un blocage sur un canal vide (présence de $\star$). Sa définition est :

$$\forall X \ \top(X) = X[\tau/\star]$$

Il faut noter que la fonction $\top$ n'est pas continue : on a $\star \prec 1$ mais pas $\tau \prec 1$.

Propriété des processus.

La propriété suivante est vérifiée par la sémantique f d'un processus[1] :

$$\vec{X} = f(\vec{X}) \Rightarrow \top(\vec{X}) \prec f(\top(\vec{X}))$$

Cette propriété à la signification suivante : si l'exécution d'un processus est maximale ($X = f(X)$) et est bloquée sur un canal vide ($X = Z\star$), alors l'exécution ne peut progresser qu'à l'instant suivant ($f(Z\tau)$).

On adopte les notations suivantes : Si f est une fonction continue et $X \prec f(X)$ alors on note $\mu_X f$ le plus petit point fixe de f, plus grand que X. On note $\bigcup X_n$ la borne supérieure d'une suite croissante X_n.

Soit un système d'équations de la forme :

$$\Sigma : \ \vec{\mathtt{X}} = \vec{\mathtt{f}}(\vec{\mathtt{X}})$$

On note $\mathcal{F}_\Sigma$ la fonction continue obtenue à partir des fonctions associées aux processus de Σ :

$$\mathcal{F}_\Sigma = \lambda \vec{X}.\vec{f}(\vec{X})$$

On définit la sémantique de Σ comme la borne supérieure $\bigcup \vec{z_n}$ de la suite $\vec{z_0}, \vec{z_1}, ...$ définie de la manière suivante :

$$\begin{aligned} \vec{z_0} &= \vec{\epsilon} \\ \vec{z}_{n+1} &= \top(\mu_{\vec{z}_n}\mathcal{F}_\Sigma) \end{aligned}$$

Cette sémantique repose sur la construction imbriquée de deux limites. La première phase de la construction de la sémantique correspond à déterminer un plus petit point fixe pour les comportements dans l'instant ($\vec{t_n} = \mu_{\vec{z}_n}\mathcal{F}_\Sigma$). A l'issue de cette première phase, tous les processus ont soit terminé leur instant, soit sont bloqués sur des canaux vides. La seconde phase correspond au

[1] On adopte une notation vectorielle pour les listes d'objets. D'autre part, on étend composante par composante la fonction $\top$ ainsi que l'ordre $\prec$.

déblocage de l'exécution ($\top(\vec{t_n})$) en forcant la terminaison de l'instant courant pour tous les processus. La sémantique du réseau est la borne supérieure ($\bigcup \vec{z_n}$) des histoires obtenues au cours des instants.

Cette définition est cohérente : Posons tout d'abord : $\vec{t_n} = \mu_{\vec{z_n}} \mathcal{F}_\Sigma$. On a :

- On a $\vec{z_0} \prec \vec{z_1}$ et t_0 est bien défini puisque $\mathcal{F}_\Sigma$ est continue.
- Par définition de $\vec{t_n}$ on a $\vec{z_n} \prec \vec{t_n}$. Puisque $\top(\vec{t_n}) = \vec{z}_{n+1}$, on a :

  $$\vec{z}_{n+1} = \vec{t_n}[\tau/\star]$$

  Donc on a : $\vec{t_n} \prec \vec{z}_{n+1}$ ce qui entraine : $\vec{z_n} \prec \vec{z}_{n+1}$.
- Puisque $\vec{t_n}$ est point fixe de $\mathcal{F}_\Sigma$, on a :

  $$\vec{t_n} = \mathcal{F}_\Sigma(\vec{t_n})$$

  Donc, par la propriété de $\mathcal{F}_\Sigma$:

  $$\top(\vec{t_n}) \prec \mathcal{F}_\Sigma(\top(\vec{t_n}))$$

  Ce qui entraine :

  $$\vec{z}_{n+1} \prec \mathcal{F}_\Sigma(\vec{z}_{n+1})$$

  Donc t_{n+1} est bien défini.

Reprenons le réseau obtenu en rebouclant la sortie du processus `Ident` sur son entrée. Le système correspondant est :

$$\Sigma : \quad \mathtt{X} = \mathtt{Ident(X)}$$

On a :

$$\begin{aligned} z_0 &= \epsilon \\ z_1 &= \top(\mu_\epsilon \mathcal{F}_\Sigma) = \top(\star) = \tau \\ z_2 &= \top(\mu_\tau \mathcal{F}_\Sigma) = \top(\tau\star) = \tau\tau \\ &\cdots \end{aligned}$$

Ainsi la sémantique du réseau est la borne supérieure des z_n c'est à dire la constante τ^ω. Dans ce réseau, $\star$ n'apparait qu'au premier instant. Ce n'est pas toujours le cas comme le montre le prochain exemple. Considérons le réseau obtenu en rebouclant la sortie sur l'entrée du processus suivant :

```
rproc Ex(in,out)
Channel in,out;
{
   int v;
   for(;;){
      Put(out,0);
      exec Get(in,&v);
      exec Delayed(in,&v);
   }
}
```

La sémantique du réseau est obtenue par :

$$z_0 = \epsilon$$
$$Ex(\epsilon) = 0\star \quad Ex(0\star) = 0\star$$
$$z_1 = 0\tau$$
$$Ex(0\tau) = 0\tau 0\star \quad Ex(0\tau 0\star) = 0\tau 0\star$$
$$z_2 = 0\tau 0\tau$$
$$Ex(0\tau 0\tau) = 0\tau 0\tau 0\star \quad Ex(0\tau 0\tau 0\star) = 0\tau 0\tau 0\star$$
$$\cdots$$

La limite des z_n est donc la suite $(0\tau)^\omega$.

Enfin, reprenons l'exemple `FLIPFLOP` du paragraphe précédent. On a :

$$Not(\epsilon) = Not(\star) = \star$$

$$Not(n \bullet X) = N(n \bullet X)$$

$$Not(\tau \bullet X) = N(\tau\tau \bullet X)$$

avec :

$$N(\epsilon) = \epsilon \quad N(\star) = \star$$

$$N(n \bullet X) = \tau \bullet N'(X)$$

$$N(\tau \bullet X) = \tau 0\tau \bullet N'(X)$$

et :

$$N'(\epsilon) = \epsilon \quad N'(\star) = \star$$

$$N'(n \bullet X) = \tau \bullet N'(X)$$

$$N'(\tau \bullet X) = N(X)$$

La sémantique $FLIPFLOP$ est construite de la façon suivante :

- $z_0 = \top(\star) = \tau$ puisque $NOT(\epsilon) = \star$ et $NOT(\star) = \star$
- On a : $u_1 = NOT(\tau) = N(\tau\tau) = \tau 0\tau \bullet N'(\tau) = \tau 0\tau$

  $u_2 = NOT(\tau 0\tau) = \tau 0\tau \bullet N(0\tau) = \tau 0\tau\tau \bullet N'(\tau) = \tau 0\tau\tau$

  $u_2 = NOT(\tau 0\tau\tau) = \tau 0\tau\tau \bullet N(\tau) = \tau 0\tau\tau\tau 0\tau$

  $u_3 = NOT(\tau 0\tau\tau\tau 0\tau) = \tau 0\tau\tau \bullet N(\tau 0\tau) = \tau 0\tau\tau\tau 0\tau\tau \bullet N(\tau) = \tau 0\tau\tau\tau 0\tau\tau\tau 0\tau$

  Ceci entaine que z_1 est $(\tau 0\tau\tau)^\omega$

Ainsi la limite des z_n est $(\tau 0\tau\tau)^\omega$ qui correspond bien à l'exécution obtenue.

5 Implémentation de Lustre et Signal en utilisant des informations d'absence

Dans cette section on implémente les langages Lustre et Signal dans les réseaux réactifs en introduisant des informations particulières d'absence. Dans cette implémentation, la structure des réseaux est figée et il ne peut y avoir plusieurs valeurs mises dans un même canal au cours d'un même instant. Les sémantiques des canaux appartiennent donc toujours au domaine $\mathcal{D}$.

Implémentation de Lustre.

On implémente Lustre en associant un processus réactif à chacun des opérateurs du langage. L'opérateur `pre` de Lustre est implémenté par :

```
rproc PRE(in,out)
Channel in, out;
{
   rauto int v;
   Put(out,FALSE);
   exec Get(in,&v);
   stop;
   for(;;){
      Put(out,v);
      exec Get(in,&v);
      stop;
   }
}
```

L'opérateur `->` de Lustre est implémenté par :

```
rproc FOL(leftIn,rightIn,out)
Channel leftIn, rightIn, out;
{
   int v;
   exec Get(leftIn,&v);
   Put(out,v);
   exec Get(rightIn,&v);
   stop;
   for(;;){
      exec Get(rightIn,&v);
      Put(out,v);
      exec Get(leftIn,&v);
      stop;
   }
}
```

En Lustre les entrées de l'opérateur `when` sont toujours définies au même instant. L'implémentation de cet opérateur est :

```
rproc WHEN(in,clk,out)
Channel in, clk, out;
{
  rauto int b, v;
  for(;;){
    exec Get(clk,&b);
    exec Get(in,&v);
    if(b==1) Put(out,v);
    else Put(out,FALSE);
    stop;
  }
}
```

L'opérateur `current` de Lustre est implémenté par :

```
rproc CURRENT(in,out)
Channel in, out;
{
  rauto int b, v, mem;
  mem = FALSE;
  for(;;){
    exec Get(in,&v);
    if(v!=FALSE) mem = v;
    Put(out,mem);
    stop;
  }
}
```

(Cette implémentation suppose en fait que la variable associée au canal d'entrée est sur l'horloge de base. Le cas général nécessite de faire apparaitre explicitement l'horloge de l'entrée.)

Considérons maintenant les extensions aux séquences des opérateurs classiques, tel l'addition. En Lustre on suppose que les arguments de ces opérateurs sont définis aux mêmes moments (ils sont sur la même horloge). Par exemple, l'extension en Lustre de la fonction "+" d'addition correspond à :

```
rproc PLUS(leftIn,rightIn,out)
Channel leftIn, rightIn, out;
{
  rauto int l, r;
  for(;;){
    exec Get(leftIn,&l);
    exec Get(rightIn,&r);
    if (l!=FALSE&&r!=FALSE)
       Put(out,l+r);
    if (l==FALSE||r==FALSE)
       Put(out,FALSE);
    if (l!=FALSE&&r==FALSE){
       for(;;){
          exec Get(rightIn,&r);
          if (r==FALSE) Put(out,r);
          else break;
       }
       Put(out,l+r);
    }else if (r!=FALSE&&l==FALSE){
       for(;;){
```

```
            exec Get(leftIn,&l);
            if (l==FALSE) Put(out,l);
            else break;
         }
         Put(out,l+r);
      }
      stop;
   }
}
```

Ainsi le n^{ieme} élément de `out` est la somme des n^{ieme} éléments de `leftIn` et `rightIn`.

Implémentation de Signal.

Le langage Signal peut également être implémenté suivant les mêmes lignes. En particulier, l'opérateur `default` est :

```
rproc DEFAULT(leftIn,rightIn,out)
Channel leftIn, rightIn, out;
{
   rauto int l, r;
   for(;;){
      exec Get(leftIn,&l);
      exec Get(rightIn,&r);
      if(l!=FALSE) Put(out,l);
      else Put(out,r);
      stop;
   }
}
```

Conclusion.

Les langages synchrones Lustre et Signal peuvent être implémentés sous forme de réseaux réactifs, de manière simple, en utilisant une information spéciale "d'absence d'information".

Accessoirement, on implémente également les extensions de Lustre et Signal dans lesquelles la récursivité est possible non seulement au niveau des variables mais également pour définir les systèmes.

L'implémentation décrite dans cette section n'est pas satisfaisante sur un point important : on est obligé de mettre une information sur un canal pour dire qu'il n'y a pas d'information sur ce canal ! Ceci pose un problème pour une exécution distribuée efficace. La question qui se pose est donc la suivante : est-il possible d'implémenter Lustre et Signal dans les réseaux réactifs, sans utiliser d'information spéciale d'absence ?

6 Implémentation de Lustre et Signal sans utiliser d'information d'absence

Dans la section 3 on a décrit l'implémentation sans utilisation d'information spéciale, des opérateurs `pre` et `->` de Lustre par les processus `Pre` et `Fol`. D'autre part, le décalage de Signal "`Y $(X)`" est équivalent à l'expression Lustre "`X -> pre(Y)`".

Bien qu'ayant le même nom, le `when` de Signal est une extension de celui de Lustre car ses arguments ne sont pas nécessairement sur la même horloge. En Signal, on peut avoir par exemple :

X	1	2		3		4	5	...
B	t	f	t			t	f	...
X when B	1					4		...

L'opérateur `when` de Signal (et donc également celui de Lustre) s'exprime en RC, sans information d'absence mais en utilisant le test de deux canaux décrit dans la section 2, de la manière suivante :

```
rproc When(in,clk,out)
Channel in, clk, out;
{
  rauto int v,b1,c,b2;
  for(;;){
    exec Delayed2(in,&b1,clk,&b2);
    if(b1==TRUE) exec Get(in,&v);
    if(b2==TRUE) exec Get(clk,&c);
    if(b1==TRUE&&b2==TRUE&&c==TRUE){
         Put(out,v);
         stop;
    }
  }
}
```

Dans cette traduction, une valeur est mise dans le canal de sortie uniquement lorsque les deux canaux d'entrée ne sont pas vides et que la valeur prise sur le canal de droite vaut vrai. Une valeur est prise à chaque fois qu'un canal d'entrée n'est pas vide. Cette implémentation est correcte car l'opérateur `when` est *strict* : quand l'une de ses entrées est absente, la sortie est également absente.

Les seuls opérateurs qu'il nous reste à traduire sont donc le `current` de Lustre et le `default` de Signal. Nous allons montrer que l'implémentation de Lustre et Signal dans les réseaux réactifs sans information d'absence équivaut à avoir le test instantané du canal vide.

Equivalence de Lustre et Signal.

On va montrer que `default` et `current` ont une puissance d'expression égale dans le sens où ils sont chacun équivalents au test instantané du canal vide.

L'opérateur `default` de Signal permet de réaliser le test instantané du canal vide, comme le montre la figure 4. Dans ce réseau, le processus `P1` produit en permanence la valeur **ff** et le processus `P2` remplace par **tt** les valeurs différentes de **ff**.

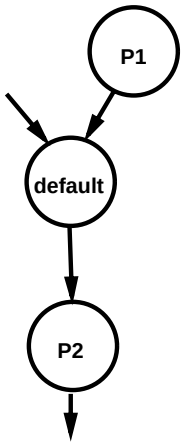

Figure 4: Test instantané du canal vide implementé avec "default"

Inversement, le test instantané du canal vide, noté "`Instantaneous`", permet de réaliser `default` par le code suivant :

```
rproc Default(leftIn,rightIn,out)
Channel leftIn, rightIn, out;
{
   rauto int l, r, vl, vl;
   for(;;){
      exec Instantaneous(leftIn,&l);
      exec Instantaneous(rightIn,&r);
      if(l==TRUE)
        exec Get(leftIn,&vl);
      if(r==TRUE)
        exec Get(rightIn,&vr);
      if(l==TRUE)
        Put(out,vl);
      else if(r==TRUE)
        Put(out,vr);
      stop;
   }
}
```

L'opérateur `default` est donc équivalent au test instantané du canal vide.

Il est clair d'autre part que le test instantané du canal vide permet de réaliser l'opérateur `current` de Lustre. L'inverse est également vrai comme le montre la figure 5 qui utilise deux processus décrits ci-dessous.

Le processus `P3` transforme son entrée en une suite strictement croissante :

```
rproc P3(in,out)
```

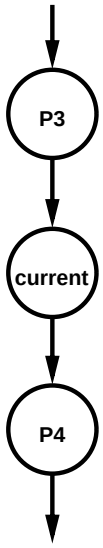

Figure 5: Test instantané du canal vide implementé avec "current"

```
Channel in, out;
{
   rauto int index;
   index = 0;
   for(;;){
     int v;
     exec Get(in,&v);
     Put(out,index++);
     stop;
   }
}
```

Le processus `P4` teste l'égalité de chaque valeur reçue avec la précédente. Si il y a égalité c'est que `current` a détecté une absence de valeur et **ff** est produit. Dans le cas contraire, **tt** est produit. Le code est le suivant :

```
rproc P4(in,out)
Channel in, out;
{
   rauto int v, mem;
   mem = 0;
   for(;;){
     exec Get(in,&v);
     if(v!=mem){
         Put(out,TRUE);
         mem = v;
     }else Put(out,FALSE);
     stop;
   }
}
```

Conclusion.

On a montré que les opérateurs `default` et `current` sont tous deux équivalents au test instantané du canal vide. Ainsi, Lustre et Signal ont tous deux une puissance équivalente à ce test.

7 Conclusion

On a défini le modèle des réseaux de processus réactifs en introduisant des points d'arrêt explicites au sein des processus séquentiels. Le test du canal vide peut être introduit dans ces réseaux sans remettre en cause leur déterminisme. On a donné une sémantique dénotationnelle des réseaux réactifs et décrit une implémentation en RC incluant le test différé du canal vide.

Les réseaux formés à partir des programmes Lustre et Signal sont des exemples de réseaux réactifs. On a décrit une implémentation de ces réseaux utilisant une information particulière d'absence. On a montré également que les opérateurs `default` de Signal et `current` de Lustre s'expriment l'un par l'autre et sont équivalents au test instantané du canal vide.

Remerciements.

Merci à Guillaume Doumenc et Robert de Simone pour leurs remarques sur une première version de ce papier.

References

[1] G. Berry, G. Gonthier, 'The Esterel Synchronous Programming Language: Design, Semantics, Implementation', INRIA Report **842** (1988), à paraitre dans *Science of Computer Programming.*

[2] F. Boussinot, 'Proposition de sémantique dénotationnelle pour des réseaux de processus avec opérateur de mélange équitable', TCS **18**, 173-206 (1982).

[3] F. Boussinot, 'Reactive C: An Extension of C to Program Reactive Systems', *Software-Practice and Experience* **21**(4), 401-428 (1991).

[4] F. Boussinot, G. Doumenc, 'Reactive C Reference Manual (version 1)', Rapport Interne ENSMP-CMA (1991).

[5] P. Caspi, 'Clocks in Dataflow Languages', Rapport LGI L-7, IMAG-Grenoble, (1988).

[6] P. Caspi, D. Pilaud, N. Halbwachs and J. Plaice, 'Lustre, a Declarative Language for Programming Synchronous Systems', *Proceedings ACM Conference on Principles of Programming Languages*, Munich (1987).

[7] G. Doumenc, F. Boussinot, 'Programmation par Objets Réactifs', Rapport Interne ENSMP-CMA (1991).

[8] G. Kahn, D. Mac Queen, 'Coroutines and Networks of Parallel Processes', Proc. IFIP 77 (1977).

[9] P. Le Guernic, A. Benveniste, P. Bournai and T. Gauthier, 'Signal: a data-flow oriented language for signal processing', *IEEE Transactions on ASSP* **34**, (2), 362-374 (1986).