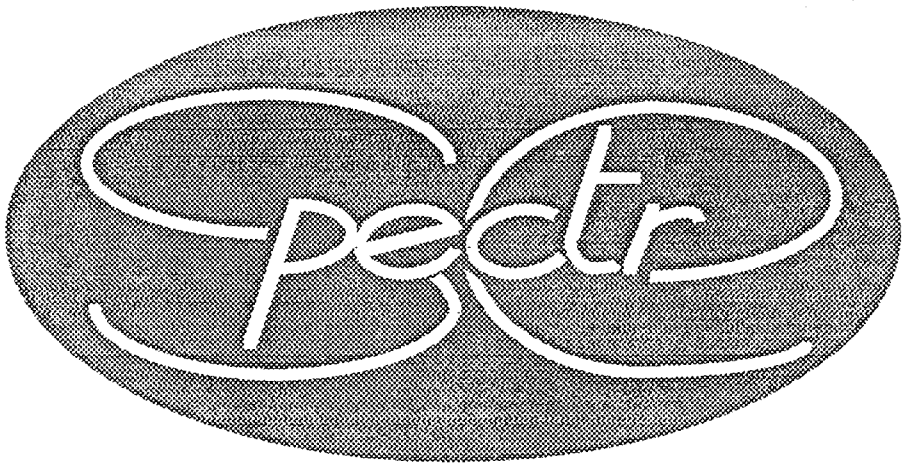
Projet SPECTRE
Groupe Spécification et Analyse des Systèmes
Laboratoire de Génie Informatique de Grenoble
IMAG-Campus, B.P.53X, 38041 Grenoble Cedex - France
Tél. +33 / 76-51-46-90
e-mail: halbwach@imag.imag.fr



L-5

Compilation séparée de Programmes LUSTRE

Pascal Raymond

juillet 88

IMAG
CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE
UNIVERSITE JOSEPH FOURIER

Sommaire

Chapitre 1
Le langage LUSTRE

La définition du langage n'a pratiquement pas changé depuis ses débuts. Cependant, chaque nouveau travail fait apparaître sa propre définition, plus par soucis de cerner quelques points essentiels que de renouveler les bases du langage. Les lecteurs intéressés sont donc invités, pour des renseignements plus complets, à se reporter aux thèses de J.-L. Bergerand [1] et de J. A. Plaice [7].

Le langage LUSTRE est un langage de style déclaratif, c'est-à-dire que les objets manipulés ne sont pas définis par la manière dont on doit les calculer, mais par des équations spécifiant leurs propriétés. En particulier la façon d'ordonner ces équations n'a aucune incidence sur la sémantique du programme.

C'est aussi un langage à flux de données, c'est-à-dire que les objets manipulés sont des suites de valeurs. La succession de ces valeurs peut être vue comme une succession d'événements, et définit donc une notion de temps.

Enfin, tout en conservant la notion de synchronisme, c'est-à-dire la simultanéité des événements, LUSTRE permet de définir, par échantillonnage, plusieurs notions de temps dans un même programme.

1.1 Variables, expressions et équations

1.1.1 Les types et les opérateurs sur les valeurs

La plupart des langages de programmation manipulent des objets de types abstraits définis à partir d'un domaine de valeurs (constantes) et d'un certain nombre d'opérateurs.

Par exemple, le domaine des booléens est défini par l'ensemble de valeurs { tt , ff } et par le comportement des opérateurs NOT, AND et OR.

Le langage LUSTRE permet d'écrire des expressions définissant des suites infinies de valeurs d'un tel type. La correspondance "type abstrait"-"suites infinies de type abstrait" se fait naturellement :

- Si c est une constante, C dénote en LUSTRE la suite : (c,c,...,c,...).
- Un opérateur op de $D_1 \times D_2 \times ... \times D_n$ dans D_{n+1}, devient en LUSTRE un opérateur sur les suites infinies de ces domaines, opérant point à point :

Si $X_i = (x_{i1},x_{i2},...,x_{ij},...)$ est une suite sur D_i, pour i = 1..n, alors

$OP(X_1,...,X_n) = (op(x_{11},...,x_{n1}), op(x_{12},...,x_{n2}),..., op(x_{1i},...,x_{ni}),...)$ est une suite sur D_{n+1}.

Par exemple, l'opérateur if-then-else sur les entiers, devient en LUSTRE un opérateur sur les suites de valeurs du domaine :

<suite de booléen> × <suite d'entier> × <suite d'entier>

dans le domaine :

<suite d'entiers>.

Les types prédéfinis en LUSTRE sont : **bool** avec les opérations booléennes, **int** (entiers relatifs) avec les opérations arithmétiques et de comparaison et **real** (nombres réels) avec les opérations arithmétiques et de comparaison. Cependant, le langage LUSTRE étant destiné à être compilé vers des langages procéduraux, la manière triviale d'étendre les types abstraits lui permet d'accepter tout type correctement défini dans le langage hôte.

1.1.2 Les opérateurs sur les suites

En plus des opérateurs sur les valeurs qui sont hérités plutôt que définis par LUSTRE, le langage possède deux opérateurs originaux de manipulation de suites.

L'opérateur unaire **pre** (mémorisation), permet de décaler une suite dans le temps, c'est-à-dire qu'à chaque instant, la valeur de **pre**(X) est la valeur de X à l'instant précédent.

Si

$X = (x_1,x_2,...,x_i,...)$,

alors

pre (X) = (nil, x_1,x_2,...,x_i,...) .

La valeur nil est la valeur indéfinie. Dans cette définition, elle exprime le fait que la valeur d'une suite avant l'instant initial n'a pas de sens.

L'opérateur binaire -> (initialisation) prend en argument deux suites de même type et renvoie la suite égale à son deuxième argument, sauf à l'instant initial où sa valeur est celle du premier.

Si

$X = (x_1,x_2,...,x_i,...)$ et $Y = (y_1,y_2,...,y_i,...)$,

alors

$X \text{ -> } Y = (x_1,y_2,...,y_i,...)$.

Ces opérateurs permettent d'écrire des équations récurrentes comme

X = 0 -> **pre** (X) + 1

qui définit X comme la suite des entiers naturels.

1.1.3 Expressions et équations

Une expression LUSTRE est un terme construit avec des constantes, des variables, des opérateurs sur les valeurs (opérateurs algébriques) et des opérateurs temporels (pour l'instant **pre** et ->). Une équation permet de définir une variable. Elle a pour forme syntaxique : Var = expression.

La sémantique intuitive d'une équation est donnée par les principes suivants :

Principe de substitution : Il y a synonymie complète entre la variable et l'expression apparaissant dans une équation, et donc, dans tout contexte on peut remplacer la variable par l'expression correspondante et réciproquement.

Principe de définition : Le comportement d'une variable X définie par une équation X = E, est complètement déterminé par cette équation et par le comportement des variables apparaissant dans l'expression E.

Dans un premier temps on va définir un programme LUSTRE comme un système d'équations de ce type. Les variables utilisées dans les expressions, mais non définies dans le système, constituent l'ensemble des paramètres du programme. Certaines variables définies dans le système d'équations sont explicitement désignées comme étant les résultats du programme.

1.2 Notion de temps

Avec les opérateurs présentés jusqu'à présent, il n'y a qu'une notion de temps, celle induite par la séquence des valeurs. Le programme a donc un comportement cyclique. Un cycle correspond à l'occurrence d'un vecteur de valeurs des suites d'entrée et, à partir de ces valeurs, au calcul des différentes variables du programme. Il est pourtant intéressant de permettre l'évolution de sous-systèmes à des rythmes différents. En effet, pour certaines applications, la valeur de certaines variables n'a de sens qu'à certains instants.

Par exemple pour un programme qui aurait pour but d'étudier une course, le calcul de la vitesse instantanée peut se faire sur la base temporelle du centième de seconde. Celui de la vitesse moyenne n'a par contre de sens que sur une base moins rapide, comme la minute, voire l'heure. Enfin d'autres calculs peuvent être envisagés sur la base du nombre de tours de circuit effectués.

On voit donc se dessiner le besoin de faire évoluer différents sous-systèmes d'un même programme à des horloges différentes. L'hypothèse de synchronisme implique que ces horloges aient une référence commune pour assurer la simultanéité d'un événement sur une base "lente" avec un événement sur une base "rapide". Sur notre exemple, cette base peut être le centième de seconde, sur laquelle sont échantillonnées les autres horloges. L'échantillonnage de l'horloge "minute" sur cette base est trivial, celui de l'horloge "tours" l'est moins. On peut en effet considérer que le passage d'un tour coïncide avec l'occurrence d'un certain centième de seconde, mais ces différents instants qui rythment les cycles de l'horloge "tours" ne se succèdent pas à intervalles réguliers, et doivent donc être dynamiquement déterminés.

On doit donc pouvoir définir en LUSTRE une horloge comme un sous-ensemble quelconque d'instants d'une horloge de référence, qu'on appellera l'horloge de base.

1.2.1 Horloges et flux

Les objets manipulés en LUSTRE, appelés flux, sont enfin définis comme étant composés :

- d'une suite de valeurs d'un type abstrait,
- d'une horloge définissant les instants où apparaissent ces valeurs.

Une horloge en LUSTRE est :

- soit l'horloge de base du programme,
- soit un flux booléen, un tel flux définit alors la suite des instants de sa propre horloge pour lesquels sa valeur est tt (Fig.1.1).

Temps de base :	*instant 1*	*instant 2*	*instant 3*	*instant 4*	*instant 5*	*instant 6*
Flux B sur la base :	**tt**	**ff**	**tt**	**tt**	**ff**	**tt**
Temps selon B :	*instant 1*		*instant 2*	*instant 3*		*instant 4*
Flux D sur B :	**tt**		**ff**	**tt**		**ff**
Temps selon D :	*instant 1*			*instant 2*		

Fig.1.1. Les flux booléens considérés comme des horloges.

1.2.2 Les opérateurs sur les horloges

L'opérateur permettant de définir l'horloge d'une expression est l'opérateur d'échantillonnage : **when**. L'expression *E when B* représente le flux, dont la suite de valeurs est la sous-suite de celles de E prises quand B vaut vrai, et dont l'horloge est B (Fig.1.2). Il est important de noter que B est alors la seule notion de temps que connait E ; en particulier la question de savoir quelle est la valeur de E quand son horloge est fausse n'a pas plus de sens que de savoir quelle valeur prend une suite entre deux indices entiers consécutifs.

E = (	e_1	e_2	e_3	e_4	e_5	e_6	...)
B = (	tt	ff	tt	tt	ff	tt	...)
E when B = (	$x_1=e_1$		$x_2=e_3$	$x_3=e_4$		$x_4=e_6$	...)

Fig.1.2. Echantillonnage.

On pose comme règle que tous les opérateurs en LUSTRE prennent comme arguments des flux ayant la même horloge. En particulier, l'expression *E + (E when B)* n'est pas une expression LUSTRE.

Pour appliquer un opérateur sur des flux ayant des horloges différentes, il faut utiliser l'opérateur de projection : **current**.

Si E est une expression d'horloge B crée par échantillonnage, l'expression current E, dénote le flux dont l'horloge est celle de B et dont la valeur à chaque instant de cette horloge est celle prise par E au dernier instant où B valait vrai (Fig.1.3.).

E = (	e_1	e_2	e_3	e_4	e_5	e_6	...)
B = (	tt	ff	tt	tt	ff	tt	...)
X = E when B = (	e_1		e_3	e_4		e_6	...)
current X = (	e_1	e_1	e_3	e_4	e_4	e_6	...)

Fig.1.3. Echantillonnage et projection.

1.2.3 Résumé

Un flux est un couple composé :

- d'une suite de valeurs d'un type algébrique,
- d'une horloge, qui est :
  - soit la base,
  - soit un flux booléen.

Les expressions LUSTRE sont composées :

- de variables flux,
- de constantes (suites infinies d'une même valeur sur la base),
- d'opérateurs sur les valeurs (opérant point à point sur des flux de même horloge),
- d'opérateurs temporels :
  - le pre (mémorisation),
  - la -> (initialisation),
  - le when (échantillonnage),
  - le current (projection).

1.3 Les nœuds

1.3.1 Nœuds et réseaux

L'abstraction procédurale en LUSTRE s'appelle le nœud. Un nœud est un opérateur sur les flux. Les opérateurs de base du langage sont donc des nœuds particuliers, et tout programme peut être vu comme un réseau de nœuds (Fig.1.4).

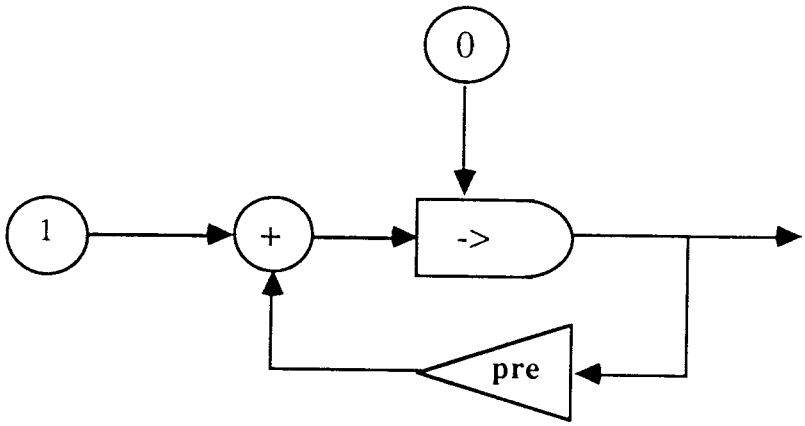


Fig.1.4 Réseau de l'équation : X = 0 –> **pre** (X) + 1.

Le programmeur peut définir en LUSTRE ses propres opérateurs avec la syntaxe :

node <nom du nœud> <entête> ; <déclarations locales> let <équations>tel .

Les déclarations locales définissent les variables locales du nœud, et des opérateurs (nœuds) non accessibles par l'extérieur.

Les équations définissent les différentes variables du nœud : sorties et flux intermédiaires.

L'entête permet de définir les paramètres,

- elle énumère les noms, types et horloges des paramètres formels d'entrée (qui sont les variables libres du système d'équations),
- elle précise quelles sont, des variables définies par le système d'équations, celles qui doivent être considérées comme les résultats du nœud, par opposition aux variables utilisées pour définir des flux intermédiaires.

L'horloge de base du nœud est par définition celle de certains paramètres d'entrée. Si on veut définir une entrée qui ne soit pas sur la base, cette horloge doit être explicitement désignée comme une nouvelle entrée du nœud. Par exemple, le nœud dont l'entête est

node MULTI-HORLOGE (hor : **bool**; x, y : **int**; (z : **int when** hor))
returns (...);

a trois entrées (qui sont par défaut) sur l'horloge de base (hor, x et y), et une entrée

échantillonnée (z sur l'entrée hor).

1.3.2 Les instanciations de nœud

Une instanciation de nœud se fait de manière fonctionnelle : le résultat d'une instanciation n'est qu'une expression particulière.

Si on définit par exemple un compteur très général comme un opérateur prenant en entrée deux flux entiers et un flux booléen et les combinant en un flux entier :

```
node COMPTEUR (init, incr : int; raz : bool)
returns (n : int);
let
  n = init -> if raz then init else pre n + incr ;
tel.
```

on peut écrire :

```
pair = COMPTEUR(0, 2, false);
mod5 = COMPTEUR(0, 1, pre(mod5)=4) );
```

ce qui définit pair comme la suite des nombres pairs, et mod5 comme la suite cyclique des entiers modulo 5.

Pour pouvoir instancier des nœuds ayant plus d'une sortie, LUSTRE permet l'utilisation de tuples. Si E_1, E_2, ..., E_n sont des expressions de types respectifs $\tau_1, \tau_2, \ldots, \tau_n$ et d'horloges $H_1, H_2, \ldots, H_n$, alors le tuple :
$(E_1, E_2, \ldots, E_n)$ est une expression de type $(\tau_1, \tau_2, \ldots, \tau_n)$ et dont l'horloge est le tuple $(H_1, H_2, \ldots, H_n)$.

De cette manière, l'instanciation du nœud pourra se faire de manière fonctionnelle : par exemple si un nœud N est défini avec l'entête

node N $(i_1 : \tau_1; i_2 : \tau_2; \ldots ; i_n : \tau_n)$
returns $(j_1 : \theta_1; j_2 : \theta_2; \ldots ; j_m : \theta_m)$;

et si $(E_1, E_2, \ldots, E_n)$ est un tuple de type $(\tau_1, \tau_2, \ldots, \tau_n)$, alors l'équation

$$(X_1,\ X_2,\ \ldots, X_m) = N\ (E_1,\ E_2,\ \ldots, E_n)$$

définit $(X_1,\ X_2,\ \ldots, X_m)$ comme un tuple de flux de type $(\theta_1,\theta_2, \ldots ,\theta_m)$.

Chapitre 2
La compilation actuelle

La compilation des programmes LUSTRE est étudiée en détail dans la thèse de doctorat de John A. PLAICE [7], qui est l'auteur du compilateur actuel. Ce chapitre présente de manière informelle un des aspects essentiels de cette compilation : la génération d'automates d'états finis.

2.1 Programmes sans blocages

Un programme LUSTRE définit l'ensemble de ses sorties comme la solution d'un système d'équations paramétré par l'ensemble de ses entrées.

La compilation ne considère que des programmes dont la solution peut être construite pas-à-pas, et dont le temps de calcul nécéssaire à chaque pas est borné. De tels programmes sont dit *sans blocages*, et se caractérisent par le fait qu'aucune variable ne dépend instantanément d'elle-même.

Typiquement, cela signifie qu'on interdit à la compilation des équations de point fixe autres que les équations récurrentes construites avec l'opérateur de retard **pre**.

En effet, une équation du type "x = f(x)", où f ne contient pas d'opérateurs de retard, n'a pas forcément de solution. Quand bien même elle en aurrait une, on ne peut pas donner de borne au temps de calcul de cette solution, ce qui viole le principe de synchronisme.

Par contre, une équation récurrente du type "x = f(pre x)" ne pose aucun problème, comme on l'a vu dans le chapitre précédent avec la suite des entiers naturels.

2.2 Un programme séquentiel simple

La correspondance "programme lustre"-"programme séquentiel" est assez naturelle. Pour cela, il faut :

• ordonner les équations LUSTRE de manière à ce qu'aucune variable ne soit utilisée avant sa définition (le fait que le programme soit sans blocage assure qu'un tel ordre existe),

• remplacer les symboles d'égalité "=" par des symboles d'affectation (qu'on note ":="),

• remplacer les opérateurs spécifiques à LUSTRE par des constructions algorithmiques, par exemple " e->e' " devient " if init then e else e' ", où init est une nouvelle variable qui devra être initialisée par le contexte appellant à vrai, puis mise à faux après l'appel.

• définir la mémoire locale en introduisant une variable P_e pour chaque "**pre** e". A la fin de chaque appel, cette variable devra être mise à jour par l'affectation : $P_e := e$.

On obtient de la sorte un programme séquentiel écrit dans un langage algorithmique "standard". La séquence des appels de ce programme correspond à l'horloge du programme LUSTRE.

Nous allons illustrer la génération de code par un exemple issu d'un système de conduite de souris. Il s'agit de déterminer le sens de rotation d'une roue (Fig.2.1), constituée de deux couronnes dont les différentes portions, alternativement opaques et transparentes, sont décalées. Deux capteurs, B_0 et B_1, situés respectivement en face des couronnes externe et interne de la roue, permettent de savoir si la portion de couronne leur faisant face est opaque ou non. Soient b0 et b1 les variables booléennes issues de ces capteurs.

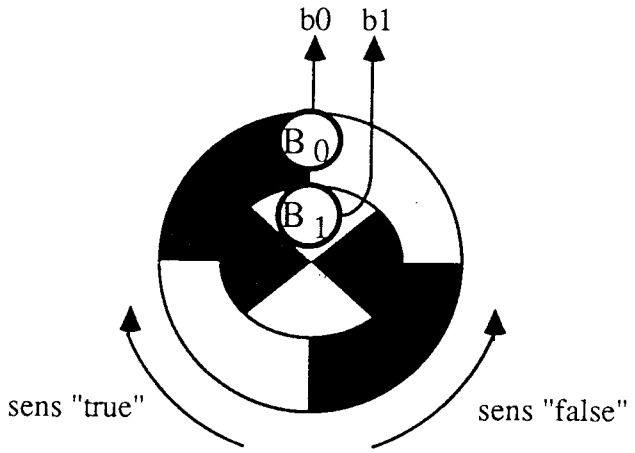


Fig.2.1. La tranche de la roue ("transparent=false", "opaque=true").

On voit sur le schéma que, quand la valeur de b0 passe de *false* à *true* (i.e. la couronne en face du capteur B_0 passe de transparent à opaque), le sens de rotation est déterminé par la valeur de b1. Le programme LUSTRE suivant compte le nombre de tour relatif d'une telle roue :

```
node SOURIS (b0, b1 : bool)  returns (rotations : int);
var sens : bool;
let
  sens = b1 when FRONT(b0);
  rotations = current(COMPTEUR(sens));
tel.

node FRONT(b : bool)  returns (front : bool);
let
  front = true -> (b and not pre(b));
 tel.

node COMPTEUR(b : bool)  returns (n : int);
let
  n = 0 -> if b then pre(n) + 1
           else pre(n) - 1;
tel.
```

La première opération consiste à "expanser" les appels de nœuds pour obtenir un programme plat. Pour cela, on doit en particulier créer des variables intermédiaires et ramener les constantes des nœuds expansés à leurs horloges effectives.

```
sens = b1 when f;
rotation = current(n);
n = 0 when f -> if sens then pre(n) + u else pre(n) - u;
u = 1 when f;
f = true -> ( b0 and not Pb0);
Pb0 = pre(b0);
```

Pour obtenir un programme séquentiel, il faut donc ordonner les équations, introduire deux variables "init-base" et "init-f" qui indiquent si on est dans l'instant inital des horloges "base" et "f". On introduit aussi une variable Pb0 pour mémoriser la valeur de l'entrée b0. Pour avoir une démarche totalement mécanique, il faudrait aussi rajouter des variables mémorisant les valeurs des expressions pre(n) et current(n) ; cet ajout est cependant inutile et on l'a négligé pour avoir un programme plus simple.

```
procedure SOURIS ( b0, b1 : bool; ref rotations : int );
var
    f, sens, Pb0 : bool;
    init-base, init-f : bool := true;

begin
    f := if init-base then true else ( b0 and not Pb0);
    if f then
            begin
                sens := b1;
                rotations := if init-f then  0
```

```
                                    else if sens then rotations + 1
                                                 else rotations - 1;
                    init-f := false;
                end;
        Pb0 := b0;
        init-base := false;
end;
```

2.3 Génération d'automates

2.3.1 Principe

Il est donc assez simple d'obtenir un code séquentiel "naïf" pour un programme LUSTRE. Cependant ce code n'est pas très efficace (on teste à chaque instant si c'est l'instant initial). Le fonctionnement d'un tel code est schématisé par la figure 2.2. La mémoire locale comporte toutes les variables correspondant aux **pre** du nœud LUSTRE et aussi la variable init. Dans cette mémoire, on s'intéresse plus particulièrement aux variables booléennes qu'on appelle variables d'état. En effet cette mémoire booléenne a un nombre fini d'états possibles. L'idée est donc d'associer à chacun de ces états un programme simplifié (où toutes les variables d'état sont remplacées par leur valeur). La phase de mémorisation des variables d'état est maintenant remplacée par la détermination de l'état suivant, auquel sera associé un autre programme simplifié.

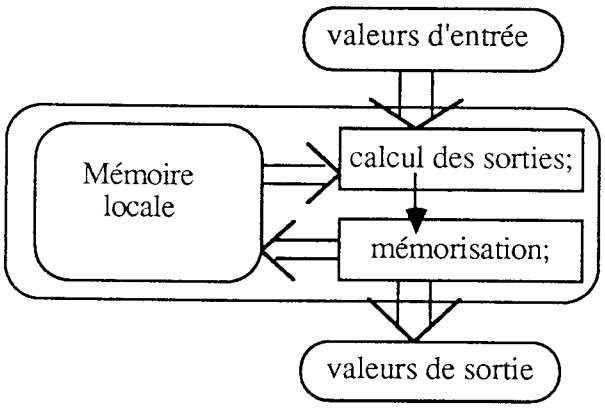


Fig.2.2 Le fonctionnement "naïf" d'un nœud LUSTRE.

2.3.2 Exemple

Pour illustrer la génération d'automate, nous reprenons l'exemple du programme SOURIS.

Les variables d'état de ce programme sont a priori : la variable Pb0, et les variables implicites init-base et init-f. En fait, la variable init-f est inutile ; un des avantage de la génération d'automate est justement d'écarter cette variable. Pour générer l'automate, on part de l'état initial où (init-base,Pb0)=(true,nil). Pour ces valeurs, le programme simplifié s'écrit :

```
sens = b1;
rotation = current(n);
n = 0;
u = 1;
Pb0 = pre(b0);
```

A partir de P_0, on peut accéder, suivant la valeur de l'entrée b0 aux états P_1 : <INIT=false,Pb0=true> et P_2 : <INIT=false,Pb0=false>, après avoir initialisé la sortie "rotation" à 0.

Les programmes correspondant à P_1 et P_2 sont :

P_1 :
```
sens = b1 when f;
rotations = current(n);
n = if sens then pre(n) + u
    else pre(n) - u;
u = 1 when f;
f = false;
Pb0 = true -> pre(b0);
```

P_2 :
```
sens = b1 when f;
rotations = current(n);
n = if sens then pre(n) + u
    else pre(n) - u;
u = 1 when f;
f = b0;
Pb0 = false -> pre(b0);
```

Dans l'état P_1, l'horloge f est fausse, donc la sortie n'est pas modifiée ; suivant la valeur de l'entrée b0, on bouclera sur P_1, ou on ira sur p_2.

Dans l'état P_2, si l'entrée b0 est fausse, on n'est pas sur un front montant, donc la sortie n'est pas modifiée et on boucle sur P_2. Par contre, si l'entrée b0 est vraie, l'horloge de n devient vraie (front montant) et la sortie doit être

modifiée : selon la valeur de l'entrée b1 il faut l'incrémenter ou la décrémenter, puis aller dans l'état P_1.

L'automate correspondant est représenté sur la figure 2.2. Les transitions sont étiquetées par leur condition et par le code séquentiel à exécuter (s'il y en a).

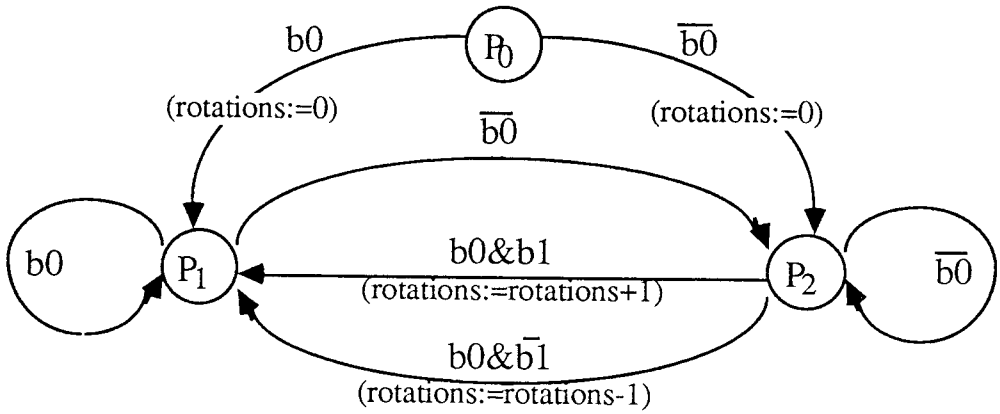


Fig.2.2 L'automate généré pour le programme SOURIS.

Chapitre 3

Le problème de la compilation séparée

L'aspect essentiel de la compilation en LUSTRE est la possibilité de générer des automates d'états finis. C'est en effet une structure rapide : le calcul des variables d'état ayant été anticipé à la compilation, le code généré pour chaque état est d'autant plus simple. Le passage du contrôle est lui aussi très simple et rapide (commutation d'état).

La contrepartie est bien évidemment l'explosion du nombre des états. En sachant que ce nombre est de l'ordre de 2^n, "n" étant le nombre de variables d'état, il est difficile d'envisager le temps de compilation ainsi que l'encombrement mémoire d'un automate réalisant une application de plusieurs centaines de variables d'état.

L'avenir de la compilation de LUSTRE doit donc passer par un compromis entre la rapidité et le moindre encombrement.

3.1 Intérêts

Jusqu'à présent, la structuration des programmes LUSTRE en réseaux de nœuds n'est pas utilisée dans la compilation. Le compilateur réalise en fait l'expansion des nœuds, c'est-à-dire, via un renommage correct, l'inclusion textuelle des équations LUSTRE correspondant à chaque instanciation de nœud.

Nous allons voir que cette méthode est la cause de la multiplication du nombre des états.

En effet, bien que la décomposition du programme en nœuds par le

programmeur puisse paraître arbitraire, elle correspond souvent à une séparation de l'ensemble des variables d'état en parties fortement indépendantes.

Si on considère par exemple un nœud A possédant n variables d'état internes (et donc potentiellement 2^n états), utilisant un nœud B à m variables indépendantes de celles de A (potentiellement 2^m états), après expansion, l'automate obtenu aura 2^{n+m} états.

Si l'on considère par contre que les deux automates sont générés indépendamment (compilation séparée), et que l'on est capable d'intégrer ces deux automates dans un système exécutable, ce système ne comportera, lui, que 2^n+2^m états.

La compilation séparée apparaît donc comme un moyen intéressant pour limiter l'explosion du nombre d'états. Il ne faut cependant pas oublier qu'il ne s'agit que d'une constatation empirique. On vient de voir ce qui se passait dans le meilleur des cas, il reste à voir ce qui se passe dans un cas plus général.

Reprenons l'exemple simple de deux nœuds A et B, possédant respectivement m et n variables d'état indépendantes, et p communes. Après expansion, le nœud obtenu possède $m+n+p$ variables d'état, soit 2^{m+n+p} états.

La compilation séparée permet d'obtenir un système à $2^{m+p}+2^{n+p}$ états. La condition pour que cette solution soit meilleure, est : $2^m+2^n \leq 2^{m+n}$, soit $m+n \geq 2$. Il suffit donc que les nœuds diffèrent sur deux variables d'état. Cette condition est suffisamment peu contraignante pour que la compilation séparée soit un moyen très intéressant de limiter l'explosion du nombre d'états.

3.2 Problèmes

La compilation séparée de la plupart des langages consiste à générer des unités de calculs (procédures, fonctions). Le cas de LUSTRE se distingue donc déjà par le fait qu'un nœud n'est pas seulement une unité de calcul, mais aussi une unité de mémorisation. En particulier, deux instanciations d'un même nœud ne sont pas deux références à un même objet, mais à des objets particuliers. Ces objets ont en commun leur schéma de fonctionnement, qui à été déterminé à la compilation, mais évoluent de manière différente.

On ne peut donc certainement pas faire correspondre aux nœuds LUSTRE des objets de type procédural simple. On pourrait cependant espérer conserver du modèle procédural la simplicité du passage des paramètres et du

contrôle (Fig.3.1.) : l'environnement appelant fournit tous les paramètres d'entrée avant de passer le contrôle à l'unité de calcul du nœud appelé. Celle-ci calcule, en fonction de ces entrées et de sa mémoire, la valeur des sorties et des variables à mémoriser (variables en PRE et état suivant), avant de rendre le contrôle au contexte appelant. Ce schéma ressemble beaucoup à la figure 2.1 ; en fait le seul changement important est que l'unité de calcul n'est pas un programme séquentiel simple, mais un automate.

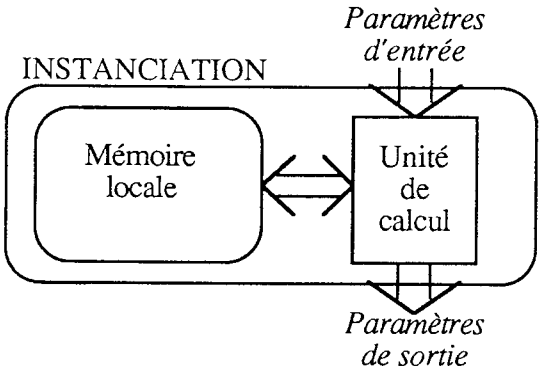


Fig.3.1. Fonctionnement "procédural" d'une instanciation de nœud LUSTRE .

Ce paragraphe s'attache à montrer que ce modèle simple n'est pas toujours valable.

3.2.1 Sémantique d'un appel de nœud

Un programme contenant une instanciation de nœud est sémantiquement équivalent au programme obtenu en substituant correctement à cette instanciation le corps du nœud correspondant.

On entend par substitution correcte une inclusion textuelle via:

- un renommage assurant la liaison paramètres formels/effectifs, et éliminant les éventuelles synonymies entre variables locales et variables de l'environnement appelant.
- l'ajout de règles explicitant les contraintes sur les horloges des paramètres effectifs. Ces contraintes étaient en effet implicites dans l'instanciation.

Les problèmes vont naître du fait que cette sémantique qu'on a voulue particulièrement naturelle, va permettre des utilisations beaucoup plus générales que celles d'un schéma procédural classique.

3.2.2 Problème de mise en séquence

L'exemple du nœud DEUX_COPIES dû à G.Gonthier [4], permet de mettre en évidence ce problème:

node DEUX_COPIES (w,x: *) **returns** (y,z:*);
let y = w ; z = x **tel**

Le code séquentiel "y:=w; z:=x", n'est pas valable pour l'appel "(A,B) = DEUX_COPIES(B,C)". En effet, la sémantique naturelle donne "A = B = C", alors que l'exécution du code donne "(A = pre(B)) & (B = C)".

On peut bien entendu imaginer le cas réciproque où la séquence "z:=x;y:=w" n'est pas valable pour l'appel "(A,B) = DEUX_COPIES(C,A)" (Fig.3.2).

La conclusion de cet exemple est que l'on ne peut pas toujours décider de la mise en séquence des calculs d'un nœud indépendamment du contexte d'appel. En fait, on peut voir intuitivement ce qui se passe. Les possibilités de mise en séquence se déduisent des dépendances entre les variables. Or, dans l'exemple de DEUX_COPIES, les variables sont séparées en deux parties indépendantes. Les contraintes propres au nœud sont donc trop lâches pour pouvoir choisir une séquence particulière.

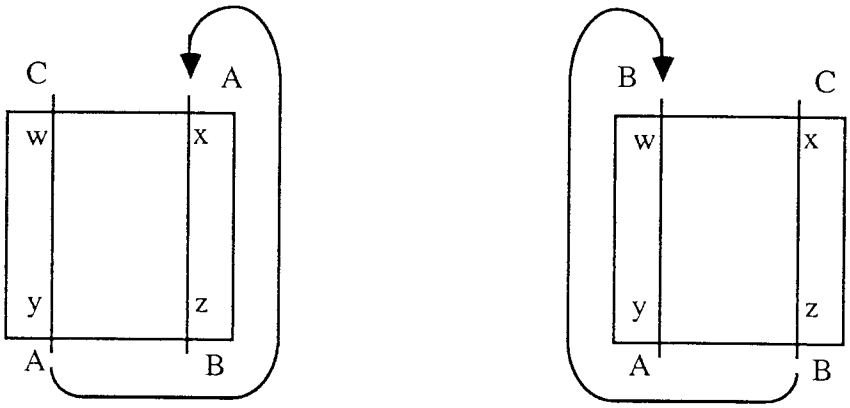


Fig.3.2. Nœud DEUX_COPIES, deux utilisations "bouclantes" possibles.

3.2.3 Problème de la communication

Pour illustrer ce problème, on introduit un exemple très proche de DEUX_COPIES:

node ET_COPIE (w,x: **bool**) **returns** (y,z:**bool**);
let y = w **and** x; z = x **tel**

Ici, le seul appel bouclant possible est "(A,B) = ET_COPIE(B,C)", car ce nœud est moins lâche que "DEUX_COPIES" sur les dépendances. Donc la séquence " z:=x; y:=w and z" semble correcte pour tout appel. Il y a cependant le problème de l'appel:
"... D = fonction-quelconque(B);... (A,B) = ET_COPIE(D,C)..."
L'opération "z:=x" va bien précéder "y:=w and z", mais ces deux opérations ne pourront pas être consécutives puisque doit s'interposer entre elles le calcul externe du paramètre effectif D.

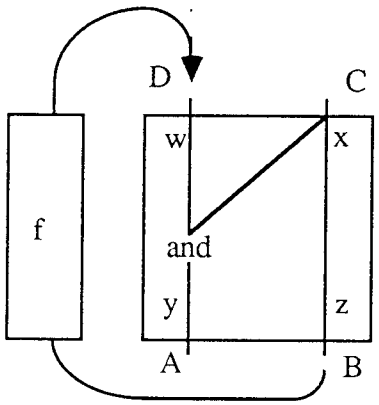


Fig.3.3. Sémantique de l'instanciation : (A = D and f(C)) & (B = C).

En fait, pour le nœud "DEUX_COPIES" on peut très bien présenter un exemple du même type. Si on introduit ce nouvel exemple, c'est pour mettre en évidence l'existence de deux problèmes qui, bien que fortement liés, sont tout de même différents:

- impossibilité d'ordonner "a priori" les opérations de n'importe quel nœud (cas de "DEUX_COPIES").
- impossibilité de fournir un code "compact" pour tout nœud (cas de "DEUX_COPIES" comme de "ET_COPIE").

3.3 Analogie avec les coroutines

Le modèle procédural ne s'appliquant pas au cas des nœuds LUSTRE, on se tourne vers le modèle plus approprié des coroutines. Dans ce modèle, le passage de contrôle entre les différentes unités de calcul est motivé par la nécessité de comuniquer des résultats intermédiaires. Il y a donc une analogie certaine avec le fonctionnement instantané du nœud ET_COPIE, dont l'exécution doit s'interrompre pour permettre le calcul du paramètre effectif D.

Pour illustrer le fonctionnement en coroutine d'une instanciation de nœud, on introduit l'exemple plus complet du nœud DEUX_BLOCS :

node DEUX_BLOCS (i_1,i_2,i_3:**bool**) **returns** (o_1,o_2:**bool**);
let o_1
= g(i_1,i_2,t) ;
z = h(i_1,i_2,t) ;
o_2 = g(t, z,i_3) ;
t = **false->pre**(z);
tel

La variable "t" et la variable implicite "init" sont les variables d'état du nœud, "g" et "h" sont des fonctions quelconques.

Pour ce nœud, on ne peut imaginer qu'un seul type d'appel bouclant : "... E = fonction-quelconque(A);... (A,B) = DEUX_BLOCS(C,D,E)...".

La figure 3.4. symbolise le fonctionnement instantané de cette instanciation.

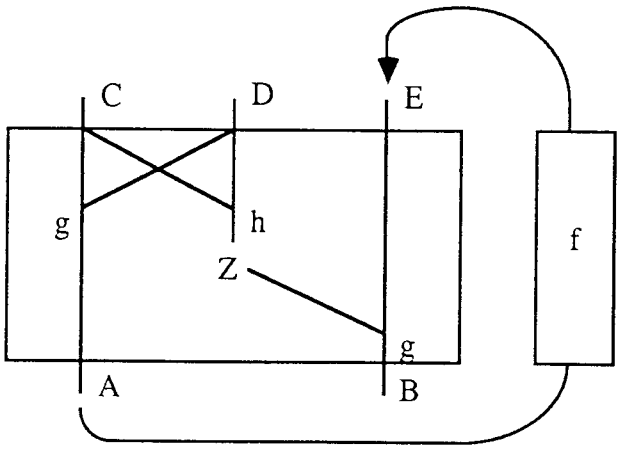


Fig.3.4. Instanciation bouclante de "DEUX_BLOCS".

Les calculs peuvent toujours se répartir, entre l'environnement appelant et le nœud, de la manière indiquée Fig.3.5. Le code y est scindé en deux parties par un point de communication, c'est-à-dire un passage de contrôle au contexte appelant, et chacune de ces parties est indivisible en ce sens que la séquence de code qui lui correspond ne nécessite jamais de communication.

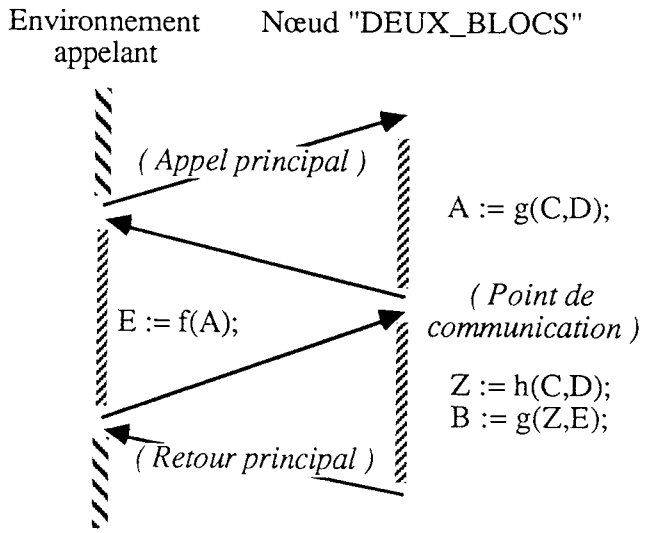


Fig.3.5. Fonctionnement en coroutine de l'instanciation bouclante.

Enfin, l'analogie avec les coroutines s'arrête, dans le cas général, au découpage en blocs de calcul indivisibles, qu'on appèle blocs procéduraux. En effet, le code d'une coroutine, bien que ponctué d'interruptions, est un code séquentiel.

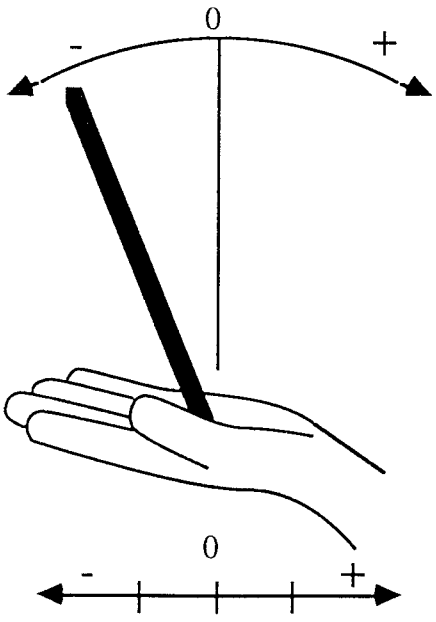
Ce n'est pas le cas, par exemple, du nœud "DEUX-COPIES", car si l'on peut scinder l'ensemble des calculs en tâches indivisibles, on ne peut ordonner ces différentes tâches indépendamment d'un contexte particulier.

3.4 Solutions

La première solution envisageable est sans doute d'ignorer les problèmes, et d'interdire les éventuelles communications entre le nœud et son environnement. Cette solution peut parraître d'autant plus raisonable que les programmes qu'on a présenté jusqu'à présent ressemble plus à des exemples d'école qu'à des problèmes réels. Pour clarifier les choses, nous allons donc présenter dans un premier temps un exemple plus concret où le problème de communication a une importance tangible.

Dans cet exemple, dit du "pendule inversé", un programme LUSTRE modélise l'interaction entre une personne et une règle posée en équilibre sur sa main (Fig.3.5).

Position relative du pendule inversé



Impulsion donnée par la main

Fig.3.5. Pendule inversé : la position du pendule est une fonction différentielle seconde de l'impulsion donnée.

Le programme LUSTRE prend en entrée l'impulsion donnée par la main, et rend en sortie la position relative du "pendule inversé". Ce qui est important dans cet exemple, c'est que la position du pendule ne dépend pas instantanément de l'entrée, mais de l'impulsion reçue deux instants auparavant.

Pour répondre aux problèmes que l'on vient d'exposer, l'idée est donc, en étudiant les possibilités de bouclage d'un nœud, de déterminer quels sont les éventuels points de communication, ou, ce qui est équivalent, de rechercher à l'interieur du nœud un ensemble de blocs procéduraux. Une première remarque évidente est que le découpage du nœud en blocs procéduraux n'est pas unique. En effet, dans l'exemple de "DEUX_BLOCS", la variable Z peut être calculée indifféremment avant ou après l'acquisition du paramètre E.

A partir d'un tel découpage, on va avoir deux types de solutions :

- une solution consiste à conserver la notion de contrôle unique, c'est-à-dire qu'on continuera à associer à tout nœud LUSTRE un objet exécutable unique. Cet objet, sera sans doute beaucoup plus compliqué que ceux qu'on génère actuellement, puisqu'il devra au moins intégrer les notions d'automate et de coroutine.
- L'autre solution consiste à répartir le contrôle, c'est à dire à associer à tout nœud un ensemble d'objets exécutables simples et indépendants.

Chapitre 4

Identification des blocs procéduraux

Le but de ce chapitre est de définir clairement quelles sont les opérations que doit effectuer un programme séquentiel modélisant le fonctionnement d'un nœud LUSTRE. Ce fonctionnement se déroule en deux temps : calcul des valeurs courantes de chacune des variables du nœud (calcul instantanés), et mémorisation des expresions en **pre** (Fig.2.1). Cette deuxième phase correpond à l'évolution du contrôle du nœud, et n'est donc pas directement liée aux problèmes de la compilation séparée. Dans le cas où l'on veut conserver la notion de contrôle unique, cette phase peut être vue comme une opération particulière et indivisible (changement d'état). Si par contre on préfère répartir le contrôle cette phase sera particulière à chacune des opérations instantanées.

4.1 Définitions

4.1.1 Opérations instantanées

On se place dans un cadre de restriction syntaxique du langage Lustre où la seule opération temporelle est **pre** et où toutes les occurrences de cette opération apparaissent dans des équations de la forme : Y = **pre** (X). Cette méthode permet de cerner de manière syntaxique les variables dont le calcul correspond à une commutation d'état, par opposition aux variables qui nécéssitent un calcul instantané.

Nous nous plaçons à un niveau d'abstraction où une opération de base

correspond au calcul d'une variable instantanée du nœud. L'ensemble des opérations de base dans lequel on va rechercher des blocs procéduraux est donc défini de la sorte :

$\mathcal{V}$ = { variables du nœud } \ { variables temporelles }

Si l'on garde l'hypothèse du contrôle unique, il suffit de rajouter à cet ensemble une variable fictive NS dont le calcul correspond à l'opération "changement d'état".

Si l'on souhaite par contre répartir le cotrôle, le calcul (i.e. la mémorisation) des variables temporelles sera réparti a posteriori, en fonction des blocs procéduraux obtenus.

On distingue dans $\mathcal{V}$ deux sous-ensembles particuliers, I et O, respectivement des variables d'entrée et de sortie,.

4.1.2 Ordre de dépendance

Les contraintes sur une séquence d'opérations correcte s'expriment en terme de dépendances entre les variables : une variable X dépend d'une variable Y, si X est définie en fonction de Y dans le programme LUSTRE. Les relations en PRE ne sont bien entendu pas prises en compte.

Cette première relation de dépendance, qu'on pourrait qualifier de "syntaxique", peut trivialement s'étendre à une relation d'ordre sur l'ensemble $\mathcal{V}$, par fermeture réflexive et transitive, la propriété d'antisymétrie étant naturellement héritée du fait que le nœud considéré est sans blocage.

On peut donc munir $\mathcal{V}$ d'un ordre $\propto \subseteq \mathcal{V} \times \mathcal{V}$ qui s'interprète par :

X $\propto$ Y si le calcul de X est nécessaire au calcul de Y.

Dans le cas du contrôle unique, l'opération NS s'interprète donc comme la dernière opération à effectuer, c'est à dire comme l'élément maximal (selon $\propto$) de $\mathcal{V}$.

4.2 Ordre de dépendance et blocs procéduraux

4.2.1 Caractérisation des nœuds "non-procéduraux"

Notre but, dans ce paragraphe, est de caractériser les nœuds posant des problèmes de mise en séquence, c'est-à-dire ceux pour lesquels on ne peut pas fournir un programme procédural simple (type Pascal) couvrant toutes les utilisations possibles du point de vue de la sémantique de Lustre. On va montrer informellement que ces nœuds sont exactement ceux qui vérifient la propriété suivante :

Définition 4.2.1: Un nœud est dit *potentiellement bouclant*, si et seulement si:

$$\exists\, i \in I, \exists\, o \in O, \neg(i \propto o)$$

Définition 4.2.1.b: Inversement, un nœud est dit *procédural* si et seulement si:

$$\forall\, i \in I, \forall\, o \in O, (i \propto o)$$

L'appellation vient du fait qu'on peut les utiliser en "bouclant" la sortie o sur l'entrée i, sans que cela introduise de blocage.

• Il est clair que, pour un tel nœud, on peut envisager une instanciation où toutes les variables du nœud ne peuvent être calculées au cours d'une même séquence d'instructions, puisque le paramètre effectif X doit être calculé exactement entre o et i. (Fig.4.1.)

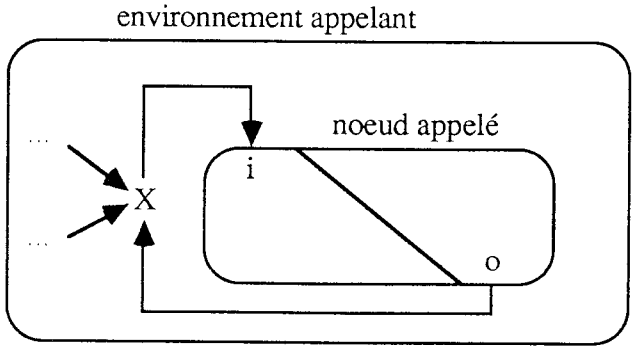


Fig.4.1. Une instanciation bouclante.

• Inversement, un nœud ne possédant pas cette propriété ne pose pas de problème; un appel correct ne pouvant alors pas être bouclant, l'environnement appelant doit être capable de fournir tous les paramètres effectifs d'entrée avant que

le nœud commence à calculer.

4.2.2 Variables compatibles

Notre but est de dégager des ensembles d'opérations, c'est-à-dire (cf.4.1.1) des ensembles de variables. Le paragraphe précédent présentait les nœuds posant des problèmes en fonction de leurs entrées et de leurs sorties. On cherche donc maintenant à se ramener à une caractérisation équivalente, mais qui porterait sur des variables quelconques.

Définition 4.2.2 : On définit deux fonctions **I** et **O** par:

$$\mathbf{I} : \mathcal{V} \rightarrow 2^{I}$$
$$x \rightarrow \{ i \in I, i \propto x \}$$

I(x) est l'ensemble des entrées nécessaires au calcul de x.

$$\mathbf{O} : \mathcal{V} \rightarrow 2^{O}$$
$$x \rightarrow \{ o \in O, x \propto o \}$$

O(x) est l'ensemble des sorties nécessitant le calcul de x.

Définition 4.2.3 : On définit sur $\mathcal{V} \times \mathcal{V}$, la relation d'incompatibilité, #, comme suit:

$$(x \# y) \Leftrightarrow$$
$$(\exists i_x \in \mathbf{I}(x), \exists o_y \in \mathbf{O}(y), \neg (i_x \propto o_y))$$
$$\text{o} \qquad \text{u}$$
$$(\exists i_y \in \mathbf{I}(y), \exists o_x \in \mathbf{O}(x), \neg (i_y \propto o_x))$$

Proposition 4.2.1:

Un nœud est potentiellement bouclant si et seulement si il possède des variables incompatibles. En fait, on a défini # pour avoir cette équivalence, la démonstration est triviale.

4.2.3 Partition des calculs en blocs procéduraux

Après cette définition, on va étudier plus en détail la relation complémentaire : χ (compatibilité).

Définition 4.2.3.b : $\chi = \mathcal{V} \times \mathcal{V} \setminus \#$

Un nœud ne contenant que des variables toutes compatibles entre elles est ce que nous avons appelé un nœud procédural. De même, à l'intérieur d'un nœud quelconque, un ensemble de variables toutes compatibles entre elles constitue un bloc de calcul ne posant pas de problème de mise en séquence. On retrouve donc la notion de "bloc procédural" introduite dans le paragraphe 3.3.2.

L'idée est de dégager dans $\mathcal{V}$ des sous-ensembles distincts de variables toutes compatibles entre elles. Il s'agit en fait de trouver une partition (c'est-à-dire une relation d'équivalence) sur $\mathcal{V}$ incluant la relation de compatibilité; de plus, on voudrait que cette équivalence soit minimale en nombre de classes, c'est-à-dire qu'à deux classes différentes corresponde bien une incompatibilité (ce que nous avions appelé "point de communication éventuel" dans le paragraphe 3.3.2.).

Proposition 4.2.2 : Une solution optimale de notre problème est une équivalence $\cong$ vérifiant:

$(x \cong y) \Rightarrow (x \chi y)$

$\neg (x \cong y) \Rightarrow (\exists x' \cong x , \exists y' \cong y , \neg (x' \chi y'))$

Ces équations indiquent simplement qu'une solution doit être <u>une</u> plus grande équivalence contenue dans χ.

Le premier écueil à éviter est que χ n'est généralement pas une équivalence:

- χ est réflexive et symétrique (évident par définition).
- χ n'est pas transitive; contre-exemple (fig. 4.2), on a bien $(z \chi o_1)$ et $(z \chi o_2)$, mais $\neg (o_1 \chi o_2)$.

Il découle directement de cet exemple que l'unicité de la solution optimale n'est pas assurée. En effet, on peut prendre indifféremment $\{i_1,i_2,o_1,z\}$ et $\{i_3,o_2,ns\}$, ou $\{i_1,i_2,o_1\}$ et $\{z,i_3,o_2,ns\}$.

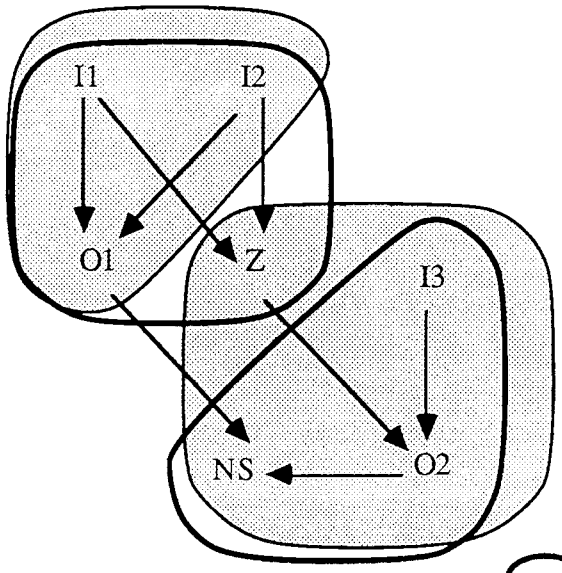


Fig.4.2.Les flèches symbolisent la relation $\propto$ .En et , les deux solutions optimales possibles.

En conclusion, on est arrivé à caractériser notre but en terme d'ordre de dépendance. Cependant, cette caractérisation (proposition 4.2.2) ne permet pas de trouver simplement une solution.

4.3 Identification des blocs procéduraux

L'impossibilité de déduire facilement de la relation de compatibilité χ une solution à notre problème, vient sans doute du fait que la relation de dépendance $\propto$ est trop contraignante. En effet, on cherchait à travers $\propto$ des informations permettant de cerner les possibilités de mise en séquence des opérations. Or, cette relation donne des informations du type "le calcul de X doit toujours être effectué avant celui de Y", là où des informations du type "le calcul de X peut toujours être effectué avant celui de Y" auraient suffi. Ce chapitre met en évidence l'existence de deux relations répondant à ce critère, et permettant de déduire deux solutions optimales.

4.3.1 Ordonnancement des entrées

Jusqu'à présent, on s'est contenté de décrire les contraintes en terme de relation de dépendance. Or, si la prise en compte de $\propto$ est nécessaire pour fournir un

programme sans blocage, elle ne suffit pas à exprimer des propriétés plus spécifiques au cas des nœuds Lustre. Cette constatation est particulièrement évidente en ce qui concerne les entrées.

En effet, la relation $\propto$ n'ordonne pas les entrées. Cependant, s'il existe deux entrées vérifiant : $\mathbf{O}(i_2) \subseteq \mathbf{O}(i_1)$, on peut dire que le calcul (i.e. l'acquisition) de i_1 doit précéder celui de i_2:

- toutes les sorties de $\mathbf{O}(i_1) \setminus \mathbf{O}(i_2)$ peuvent être rebouclées sur i_2, et, dans ce cas, on devra les calculer avant i_2 (et forcément après i_1).
- par contre, le problème ne se pose pas pour les sorties de $\mathbf{O}(i_2)$, qui ne peuvent pas être rebouclées sur i_1.

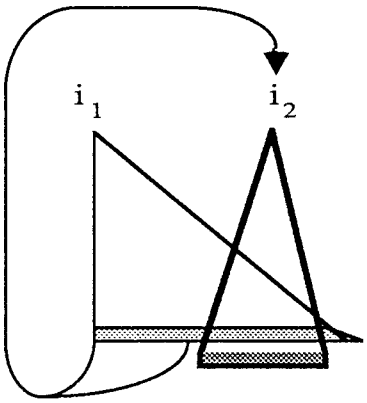


Fig.4.3.a.

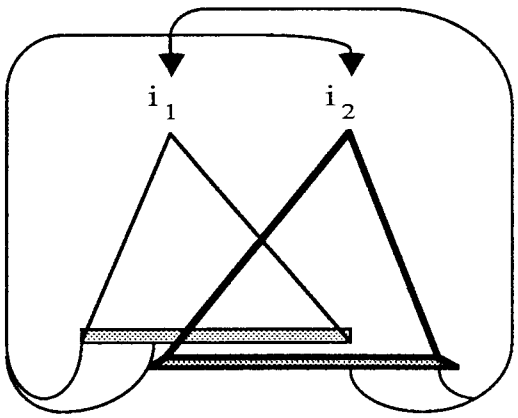


Fig.4.3.b

S'il y a inclusion, (Fig.4.3.a.), seules des sorties de i_1 peuvent être rebouclées, donc i_1 peut toujours être calculée en premier. Par contre, si on n'a pas l'inclusion (Fig.4.3.b), des sorties de i_1, comme de i_2, peuvent être rebouclées. On ne peut donc pas décider localement de la mise en ordre de ces deux variables; seule l'utilisation du nœud permettra de le faire (cas typique de variables incompatibles).

4.3.2 Relations de précédence

Dans l'exemple précédent, l'intérêt de prendre en compte la relation "$\mathbf{O}(y) \subseteq \mathbf{O}(x)$" est particulièrement évident car cela permet d'ordonner les entrées, alors que la relation de dépendance ne le pouvait naturellement pas. Mais les mêmes arguments peuvent être repris pour des variables quelconques, en effet :

- si deux variables quelconques sont dépendantes, par exemple $x \propto y$, on a

naturellement : $\mathbf{O}(y) \subseteq \mathbf{O}(x)$. La prise en compte de cette nouvelle relation couvre donc l'ordre de dépendance.

• si par contre elles ne le sont pas (Fig.4.4.a), les arguments de l'exemple restent valables : seules des sorties de x peuvent être rebouclées, et on peut dire que x peut toujours être calculée avant y.

Un raisonnement similaire peut être tenu avec la relation "$\mathbf{I}(x) \subseteq \mathbf{I}(y)$". Dans ce cas aussi (Fig.4.4.b), seules des sorties de x peuvent être rebouclées et on peut donc en déduire que x peut toujours être calculée avant y.

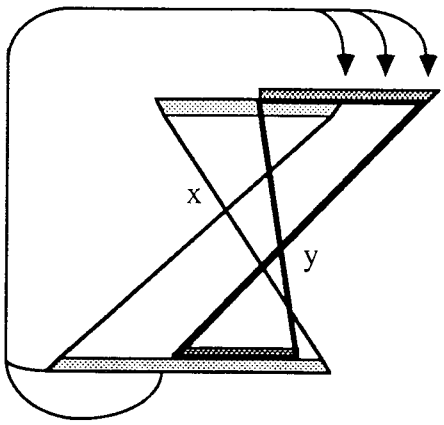


Fig.4.4.a

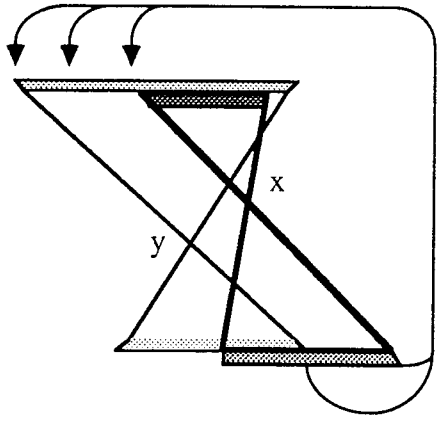


Fig.4.4.b

On va définir, à partir de $\propto$ et de $\mathbf{O}$, une nouvelle relation, la précédence par rapport aux sorties, notée $\propto_o$, et dualement, une relation de précédence par rapport aux entrées, définie à partir de $\propto$ et de $\mathbf{I}$, et notée $\propto_i$.

Définition 4.3.1:

$\forall x \in \mathcal{V}$, $\forall y \in \mathcal{V}$,

$(x \propto_o y) \Leftrightarrow (\mathbf{O}(y) \subseteq \mathbf{O}(x))$

$(x \propto_i y) \Leftrightarrow (\mathbf{I}(x) \subseteq \mathbf{I}(y))$

Propriété 4.3.1: $\propto_o$ et $\propto_i$ sont des préordres. Evident par définition : elles conservent toutes les propriétés de l'inclusion, sauf l'antisymétrie, puisque ni $\mathbf{O}$ ni $\mathbf{I}$ ne sont surjectives.

4.3.3 Propriété de bouclage et compatibilité

Les relations de précédence, $\propto_o$ et $\propto_i$, permettent donc de compléter les informations sur les ordonnancements possibles des calculs. Mais il est important de

noter qu'elles n'ajoutent aucune relation entre entrées et sorties par rapport à la relation de dépendance.

Propriété 4.3.2: $\forall i \in I, \forall o \in O, (i \propto_i o) \Leftrightarrow (i \propto_o o) \Leftrightarrow (i \propto o)$
en effet: • $(i \propto o) \Rightarrow (i \propto_o o)$ et $(i \propto o) \Rightarrow (i \propto_i o)$, évident par définition.

• $(i \propto_o o) \Leftrightarrow \mathbf{O}(o) \subseteq \mathbf{O}(i)$, en particulier, $o \in \mathbf{O}(o)$, donc $o \in \mathbf{O}(i)$, soit $(i \propto o)$.

• $(i \propto_i o) \Leftrightarrow \mathbf{I}(i) \subseteq \mathbf{I}(o)$, en particulier, $i \in \mathbf{I}(i)$, donc $i \in \mathbf{I}(o)$, soit $(i \propto o)$.

En particulier, on peut remplacer dans la définition des nœuds non-procéduraux (définition 3.2.1), la relation $\propto$ par $\propto_o$ ou par $\propto_i$, et reprendre un raisonnement similaire sur les variables compatibles.

Définition 4.3.2: Les fonctions $\mathbf{I}_o$, $\mathbf{O}_o$, $\mathbf{I}_i$ et $\mathbf{O}_i$ sont définies par:

$\mathbf{I}_o$: $\mathcal{V} \rightarrow 2^I$ $\quad x \rightarrow \{ i \in I, i \propto_o x \}$ $\qquad$ $\mathbf{O}_o$: $\mathcal{V} \rightarrow 2^O$ $\quad x \rightarrow \{ o \in O, x \propto_o o \}$

Et de manière duale:

$\mathbf{I}_i$: $\mathcal{V} \rightarrow 2^I$ $\quad x \rightarrow \{ i \in I, i \propto_i x \}$ $\qquad$ $\mathbf{O}_i$: $\mathcal{V} \rightarrow 2^O$ $\quad x \rightarrow \{ o \in O, x \propto_i o \}$

Définition 4.3.3 : Compatibilité vis-à-vis des sorties.

$(x \chi_o y) \Leftrightarrow (\forall i_x \in \mathbf{I}_o(x), \forall o_y \in \mathbf{O}_o(y), (i_x \propto_o o_y))$
et $(\forall i_y \in \mathbf{I}_o(y), \forall o_x \in \mathbf{O}_o(x), (i_y \propto_o o_x))$

Et de manière duale: Compatibilité vis-à-vis des entrées.

$(x \chi_i y) \Leftrightarrow (\forall i_x \in \mathbf{I}_i(x), \forall o_y \in \mathbf{O}_i(y), (i_x \propto_i o_y))$
et $(\forall i_y \in \mathbf{I}_i(y), \forall o_x \in \mathbf{O}_i(x), (i_y \propto_i o_x))$

Propriété 4.3.3: On a : $\mathbf{O}_o = \mathbf{O}$ et $\mathbf{I}_i = \mathbf{I}$
en effet:
• $\mathbf{O}_o(x) \subseteq \mathbf{O}(x)$ et $\mathbf{I}(x) \subseteq \mathbf{I}_i(x)$, évident par définition.
• $o \in \mathbf{O}(x) \Rightarrow (x \propto o) \Rightarrow \mathbf{O}(o) \subseteq \mathbf{O}(x) \Rightarrow (x \propto_o o) \Rightarrow o \in \mathbf{O}_o(x)$
• $i \in \mathbf{I}(x) \Rightarrow (i \propto x) \Rightarrow \mathbf{I}(i) \subseteq \mathbf{I}(x) \Rightarrow (i \propto_i x) \Rightarrow i \in \mathbf{I}_i(x)$

En faisant intervenir la définition des relations de précédence ainsi que la propriété 4.3.3, on obtient la définition équivalente :

Définition 4.3.3.bis : Compatibilité vis-à-vis des sorties.

$$(x \chi_o y) \Leftrightarrow (\forall i \in \mathbf{I}_o (x) , \mathbf{O}(y) \subseteq \mathbf{O}(i)) \text{ et } (\forall i \in \mathbf{I}_o (y) , \mathbf{O}(x) \subseteq \mathbf{O}(i))$$

Et de manière duale: Compatibilité vis-à-vis des entrées.

$$(x \chi_i y) \Leftrightarrow (\forall o \in \mathbf{O}_i (x) , \mathbf{I}(o) \subseteq \mathbf{O}(y)) \text{ et } (\forall o \in \mathbf{O}_i (y) , \mathbf{I}(o) \subseteq \mathbf{O}(x))$$

Comme dans la première partie de ce chapitre, ces définitions ont été choisies pour avoir trivialement l'équivalence :

"Le nœud est non-procédural $\Leftrightarrow$ il possède des variables incompatibles".

En reprenant l'exemple du nœud DEUX_BLOCS, on voit immédiatement ce qu'apportent ces deux relations par rapport la compatibilité χ. Dans cet exemple, on a vu que la variable Z est indifféremment compatible (selon χ) avec les variables O_1 et O_2, alors que ces deux dernières ne sont pas compatibles entres elles. Avec la relation χ_i, Z est toujours compatible avec O_1, mais ne l'est plus avec O_2. Inversement, selon χ_o, Z est compatible avec O_2, mais pas avec O_1. Sur cet exemple, ces deux relations ont donc l'avantage d'être transitives contrairement à χ.

4.3.4 Data-équivalence et demand-équivalence

Le fait que la propriété de bouclage soit équivalente pour $\propto$, $\propto_o$ et $\propto_i$ nous permet de reprendre la propriété 3.2.2 (caractérisation des solutions optimales) en remplaçant indifféremment χ par χ_i ou χ_o. Cependant, alors que le système initial n'offre pas de solution évidente, les deux nouveaux font apparaître chacun une solution triviale.

Propriété 4.3.4 : La compatibilité vis-à-vis des sorties est une équivalence qui s'exprime par: $(x \chi_o y) \Leftrightarrow (\mathbf{I}_o (x) = \mathbf{I}_o (y))$

Et de manière duale: La compatibilité vis-à-vis des entrées est une équivalence qui s'exprime par: $(x \chi_i y) \Leftrightarrow (O_i (x) = O_i (y))$

En effet: $(x \chi_o y) \Leftrightarrow (\mathbf{I}_o (x) = \mathbf{I}_o (y))$, car

$\Leftarrow$ évident,

$\Rightarrow$ •Soit $i \in \mathbf{I}_o (x)$,

$\mathbf{O} (y) \subseteq \mathbf{O} (i)$ (définition 5.2 bis)

$(i \propto_o y)$, (définition 5.1)

$(i \in \mathbf{I}_o (y))$, (définition 5.3)

•Et symétriquement pour $i \in \mathbf{I}_o (y)$.

La démonstration pour χ_i est similaire.

Ces deux solutions, χ_i et χ_o, sont a priori différentes; si on reprend l'exemple de la figure 3.2, on retrouve exactement, en gris les classes de χ_o, et en clair celles de χ_i.

En fait, on retrouve ici, de manière assez intuitive, la dualité Demand-driven / Data-driven qui est une constante des problèmes de compilation.

• χ_o et "Demand-driven" (ne calculer que ce qui est nécessaire) :

Dans l'exemple, $Z \chi_o O_2$, s'interprète par : le calcul de Z est compatible avec celui de O_2 (aspect compilation séparée) et, quand on doit calculer O_2, on doit calculer Z (aspect "Demand-driven").

• χ_i et "Data-driven" (calculer tout ce qu'il est possible de calculer):

Dans l'exemple, $Z \chi_i O_1$, s'interprète par : le calcul de Z est compatible avec celui de O_1 (aspect compilation séparée) et, quand on peut calculer O_1, on peut calculer Z (aspect "Data-driven").

Chapitre 5

Les réseaux de nœuds procéduraux

Nous avons vu (§ 3.4) qu'il y a deux types de solution pour exploiter le découpage des calculs d'un nœud en blocs procéduraux. Une des solutions possibles consiste à associer à tout nœud un objet unique. Cet objet, comme on l'a vu dans le chapitre 3, doit concilier les notions d'automate et de coroutine : c'est donc à priori un objet complexe, d'autant plus que le modèle de fonctionnement en coroutine est limité (§ 3.3.2). On est donc loin de pouvoir garantir qu'un tel objet réponde au critère de fiabilité. Inversement, la solution retenue, qui sous entend la repartition du contrôle entre les différents blocs procéduraux d'un nœud, permet d'obtenir un ensemble d'objets simples, semblables aux automates générés actuellement.

5.1 Principe

Dans le chapitre précédent, nous avons caractérisé les nœuds procéduraux, c'est-à-dire ceux pouvant être utilisés par n'importe quel contexte comme une simple procédure. Tout nœud peut être vu comme un réseau de nœuds de ce type. Pour s'en convaincre, il suffit de considérer un nœud comme le réseau de tous ses opérateurs de base. En effet, on a vu (§ 1.3.1) qu'un opérateur est un nœud particulier, et trivialement procédural (puisque sa seule sortie dépend de toutes ses entrées).

La figure 5.1 représente le nœud DEUX_BLOCS du chapitre 3 sous la forme d'un réseau. Le nœud correspondant à l'opérateur -> n'est pas représenté car il ne joue pas un rôle déterminant : tout se passe comme si l'opérateur **pre** renvoyait

"false" au lieu de "nil" à l'instant initial.

Bien évidemment, associer à tout opérateur un automate ne constitue pas une solution acceptable. L'intérêt de la génération d'automate est justement de pouvoir réunir des calculs autour d'une structure de contrôle commune, déduite de la mémoire booléenne.

L'idée est donc d'obtenir pour tout nœud un découpage optimal, assez fin pour ne faire apparaître que des nœuds procéduraux, mais pas trop pour pouvoir générer des automates intéressants. On retrouve donc tout à fait la notion de blocs procéduraux du chapitre précédent.

Pour reprendre notre exemple, on a vu dans le chapitre 4, grâce à la relation de compatibilité $\cong_O$, que les variables z et O_2 peuvent toujours être calculées ensembles. On peut donc chercher à réunir ces deux calculs dans un même sous-nœud de DEUX_BLOCS.

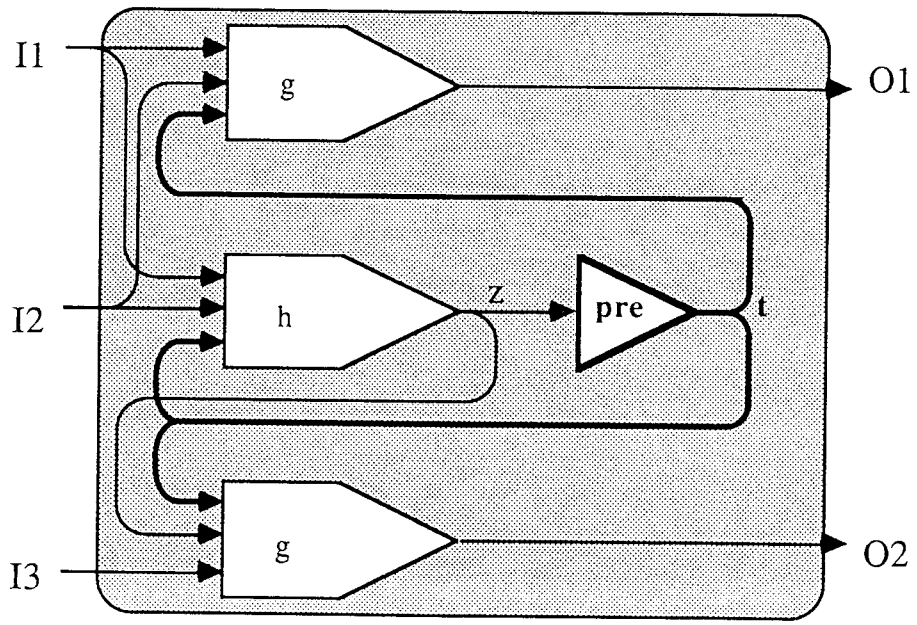


Fig 5.1 Le réseau simplifié du nœud DEUX_BLOCS

5.2 Synthèse d'un nœud procédural

Notre but va être d'associer à chacune des classes de $\mathcal{V}/\cong_O$ un nœud procédural. Soit $C_k = \{ X_{k1}, \ldots, X_{kn} \}$ une de ces classes ; le résultat espéré est

le nœud procédural "le plus simple" comportant les équations :

$X_{k1} = E_{k1}; \ldots; X_{kn} = E_{kn}$

qui apparaissaient dans le nœud initial.

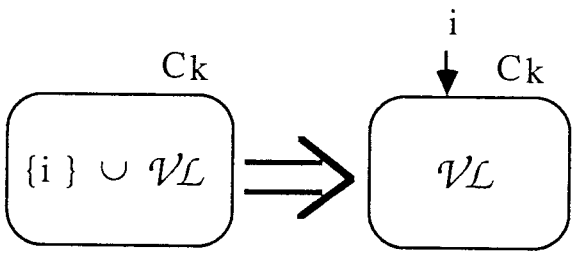
Dans notre exemple, on aura donc deux sous-nœuds,

C_1 comportant l' équation : $o_1=g(i_1, i_2, t)$, et

C_2 comportant les équations : $z=h(i_1, i_2, t)$; $o_2=g(t, z, i_2)$.

Il est clair qu'il faut au moins ajouter à un tel système un entête comportant les variables de sortie qui y apparaissent. Mais il ne suffit pas d'y ajouter les variables d'entrée car ce système peut comporter d'autres variables libres que celles du réseau total. Les variables libres d'un tel système peuvent être de plusieurs types, nécessitant chacun une étude particulière. Pour chacun de ces cas on schématise le traitement à effectuer par une figure : la partie gauche symbolise le nœud avant traitement, la partie droite après ; dans chaque cas, le traitement fait disparaître une variable libre du nœud.

• Il peut s'agir évidemment d'une variable libre de tout le réseau, c'est à dire une variable i de I (entrées du réseau), auquel cas elle doit être définie comme une entrée du nœud. La figure ci-dessous symbolise ce cas, $\mathcal{VL}$ est l'ensemble des variables libres qui restent à considérer :



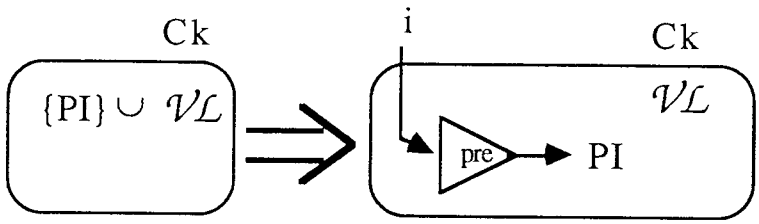
Pour notre exemple, on a au moins :

C_1 (i_1, i_2 : bool; ...) returns (O_1 : bool; ...) et

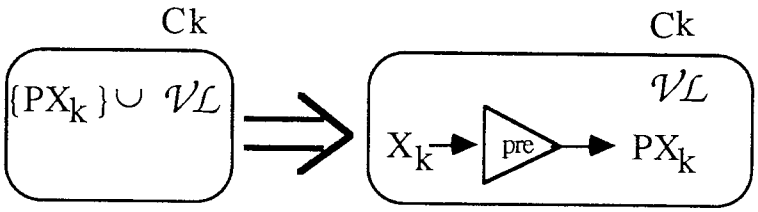
C_2 (i_1, i_2, i_3 : bool; ...) returns (O_2 : bool; ...).

• S'il s'agit d'une variable Pi définie dans le nœud initial par une équation Pi = pre i, où i ∈ I, on peut toujours rajouter cette équation à notre système sans que cela pose de problème. Si elle ne l'est pas déjà, i devient alors une entrée du

nœud.

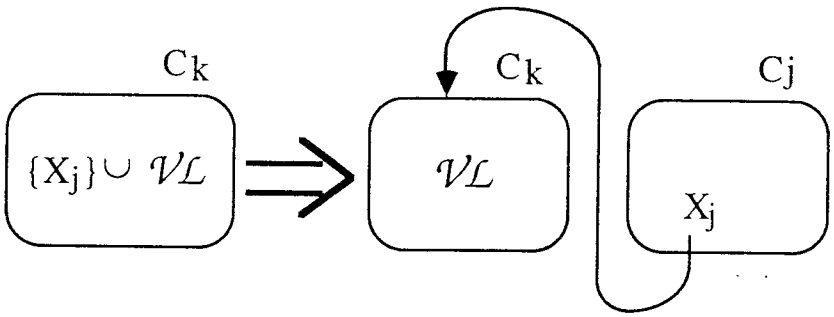


• De même pour une variable PX_{kn} = pre X_{kn}, où $X_{kn} \in C_k$, cette équation peut être rajoutée au système sans que cela modifie l'aspect procédural du nœud. Avec les variables en PI, ces variables constituent la mémoire locale du nœud autour de laquelle est synthétisé le contrôle de l'automate correspondant.



Dans l'exemple, on peut donc rajouter au système d'équations de C_2 la définition t = **pre** z.

• Pour une variable instantanée X_{jn} définie dans une autre classe C_j, c'est-à-dire une variable incompatible avec celles de C_k, il n'est pas question d'ajouter l'équation $X_{jn} = E_{jn}$. Une variable de ce type va être considérée comme une nouvelle entrée du nœud et de manière duale, le nœud correspondant à la classe C_j aura une nouvelle sortie X_{jn}.



• Il reste enfin le cas des variables du type PX_{jn} = pre X_{jn}, où $j \neq k$.

Une solution consiste à ajouter cette équation au système, et définir (si

ce n'est déjà fait) X_{jn} comme une entrée du nœud. Cette méthode a l'avantage d'augmenter le nombre de variables de contrôle, et donc d'affiner le découpage en états de l'automate correspondant, qui peut paraître sommaire si on se contente des variables temporelles locales. Mais cet ajout n'est pas toujours possible. En effet, une relation d'ordre sur les classes d'équivalence de $\mathcal{V}/\equiv_o$ se déduit de la relation de précédence sur les variables .

Cet ordre est défini par : $\forall\ C_1, C_2 \in \mathcal{V}/\equiv_o$

$$(C_1 \propto_o C2) \Leftrightarrow (\exists\ x_1 \in C_1,\ \exists\ x_2 \in C_2\ x_1 \propto_o x_2)$$

Il est clair que procéder comme on vient de le dire va ajouter des dépendances entre les classes. Or ces nouvelles dépendances découlent de l'opérateur **pre**, c'est-à-dire qu'elles ne sont pas prises en compte par $\propto_o$, d'où le risque d'avoir un blocage.

C'est le cas du nœud DEUX_BLOCS (Fig.5.2), on a $C_1 \propto_o C_2$, car $o_1 \propto_o o_2$, cette relation est représentée par la flèche épaisse. Définir z comme une entrée instantanée du nœud crée une dépendance incompatible avec cet ordre.

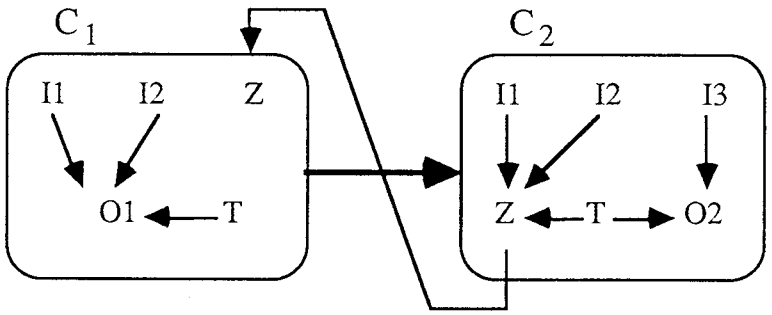


Fig.5.2 DEUX_BLOCS : 1ère répartition blocante des calculs temporels.

Une autre possibilité consiste à déclarer non plus X_{jn}, mais PX_{jn} comme entrée du nœud C_k et comme sortie du nœud C_j. Mais là encore le blocage est possible, comme on le voit sur notre exemple (Fig.5.3).

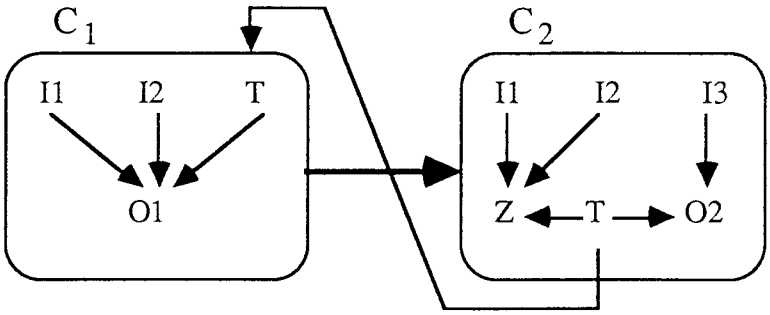


Fig.5.3 DEUX_BLOCS : 2ème répartition blocante des calculs temporels.

Il y a en fait deux cas de variable temporelle échangée :

• Si le bloc d'origine de la variable PX_{jn} (C_j), précède le bloc utilisateur (C_k) dans l'ordre des classes considéré, on peut indifféremment accepter en entrée de C_k la variable PX_{jn} devenue une sortie de C_j, ou alors définir X_{jn} comme la variable échangée et rajouter un opérateur **pre** dans le nœud C_k. C'est en particulier le cas où X_{jn} est déjà une entrée du nœud.

• Si par contre les deux blocs sont ordonnés dans l'autre sens, ou si ils ne sont pas du tout ordonnés, la seule solution consiste à définir un nouveau nœud **pre** à l'extérieur des deux autres. L'exemple précédent nous l'a montré dans le cas de blocs ordonnés, mais les mêmes arguments sont valable pour deux blocs incomparables. En effet, l'absence de relation de précédence entre deux blocs peut s'interpréter comme une dépendance possible dans un sens comme dans l'autre (nœud DEUX_COPIES §3.2.2).

5.3 Le réseau

On peut donc, grâce aux équivalences sur les variables instantanées, associer à tout nœud un réseau optimal, en ce sens que deux sous–nœud procéduraux ne peuvent être fusionnés en un nœud qui soit lui-même procédural.

Dans notre exemple du nœud DEUX_BLOCS, le réseau obtenu comporte trois nœuds procéduraux : C_1, C_2 et C_T (Fig.5.4).

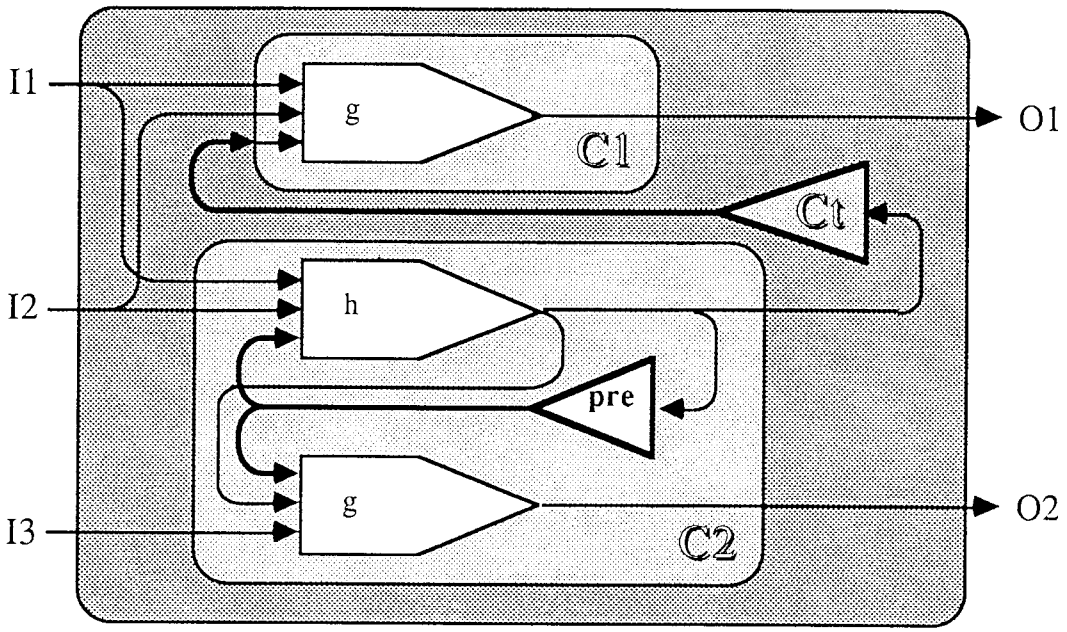


Fig.5.4 Réseau optimal pour le nœud DEUX_BLOCS (demand-équivalence).

La figure 5.5 montre l'autre solution, obtenue à partir de la data-équivalence. Cet exemple est aussi intéressant, car on y voit une variable instantanée échangée entre deux blocs. La dépendance engendrée ne crée pas de blocage, car elle correspond à une précédence déjà existante. Dans ce cas précis, le nœud C_t n'est donc pas nécessaire. On peut supprimer cet opérateur et accepter directement t comme entrée de C_2. Mais on peut aussi définir z comme entrée et intégrer le nœud **pre** dans C_2.

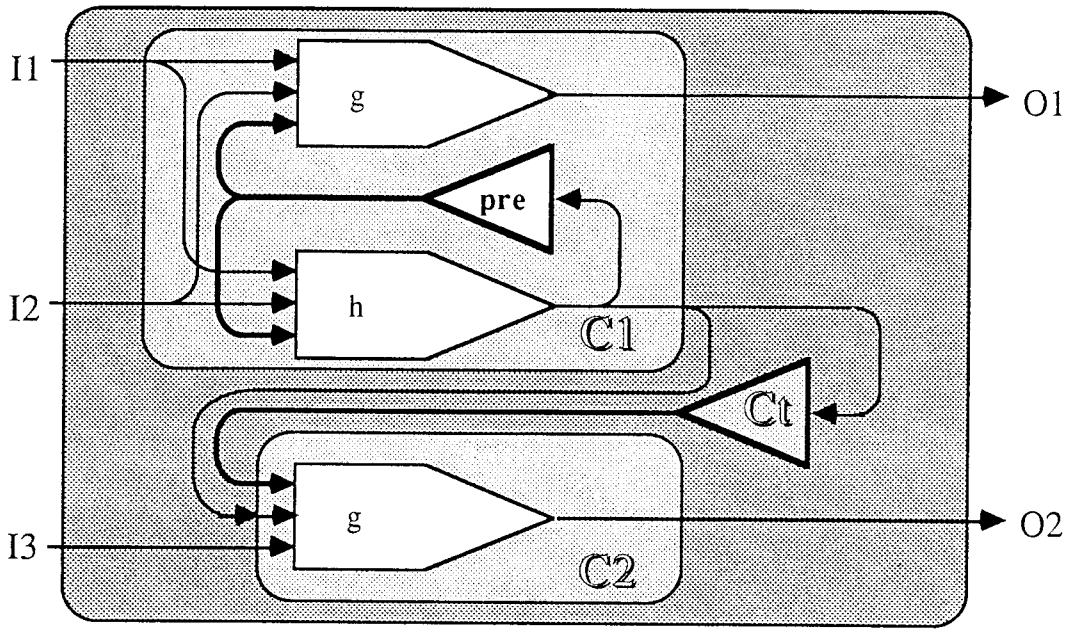


Fig.5.5 Réseau optimal pour le nœud DEUX_BLOCS (data-équivalence).

5.4 Application à la compilation

L'avantage essentiel de la solution présentée dans ce chapitre est que le produit obtenu peut s'exprimer simplement sous la forme d'un programme LUSTRE. Après analyse suivant la demand-équivalence, le nœud DEUX_BLOCS du chapitre 3 se réécrit sous la forme suivante :

node *DEUX_BLOCS* (i_1, i_2, i_3 : **bool**) **returns** (o_1, o_2 : **bool**);
let o_1 = C1(i_1, i_2,t) ;
(z, o_2) = C2(i_1, i_2, i_3) ;
t = **false->pre**(z)
tel.

Où les sous-nœuds C1 et C2 sont définis par :

node C1 (x, y, u : **bool**) **returns** (v : **bool**);
let v = g(x, y, u);
tel;

```
node C2 ( x, y, u : bool ) returns ( v, w : bool );
let   v = h(x, y, c ) ;
      w = g( c, v, u ) ;
      c = false->pre(v)
tel;
```

Ces deux derniers peuvent alors être compilés en automates d'états finis. Au niveau du nœud principal, on peut donc considérer que C_1 et C_2 sont de simples procédures externes, comme l'étaient f et g dans le nœud originel. Bien entendu, le nouveau corps du nœud DEUX_BLOCS devra être expansé dans le contexte utilisateur, car ce n'est qu'à ce niveau qu'on pourra décider d'un ordre des appels sans blocage. Mais cette expansion est maintenant réduite à ce qu'il est réellement impossible de déterminer quand on ne connait pas le contexte d'utilisation du nœud.

5.4.1 La phase d'analyse

Un compilateur basé le découpage en réseau doit donc manipuler trois types de nœuds LUSTRE :

- les nœuds écrits par l'utilisateur (par exemple le nœud DEUX-BLOCS originel),
- les nœuds procéduraux déduits à la compilation (C_1 et C_2), auxquels on peut associer un automate,
- les nœuds de type "réseau", qu'on appelera schémas d'utilisation, sont ce qui reste du programme après analyse. Comme *DEUX-BLOCS*, ces nœuds ne contiennent que des appels de fonctions et éventuellement des points de mémorisation. Par exemple, le schéma d'utilisation d'un nœud écrit par l'utilisateur, qui serait déjà procédural, se réduit à un seul appel de fonction.

La figure 5.6. représente l'architecture d'un compilateur LUSTRE basé sur ces principes. Ce compilateur est constitué de deux couches distincte. La première, constituée de l'analyseur, ne manipule que des nœuds LUSTRE. Le travail de cet analyseur consiste en :

- l'analyse statique du nœud à compiler (vérification des types, calcul des horloges),
- l'expansion des schémas d'utilisation des sous-nœuds (ceux-ci ont été compilés au préalable),

• enfin, la synthèse d'un réseau de nœuds procéduraux équivalent, sous la forme d'un schéma d'utilisation (qui peut être réutilisé par l'analyseur) et d'un ensemble de nœuds procéduraux (qui peuvent être compilés en automates d'états finis).

5.4.2 La génération de code

La deuxième phase consiste en la génération de code proprement dite. Pour des raisons de portabilité, le générateur d'automate produit des programmes OC (*format portable LUSTRE-ESTEREL*). Le langage OC est un formalisme de description d'automates commun aux langages LUSTRE et ESTEREL [6]. De cette manière, les compilateurs de ces deux langages peuvent utiliser les mêmes générateur de code exécutable (compilateurs OC). Actuellement, on utilise un générateur de programmes C, mis au point à Sophia-Antipolis dans l'équipe ESTEREL.

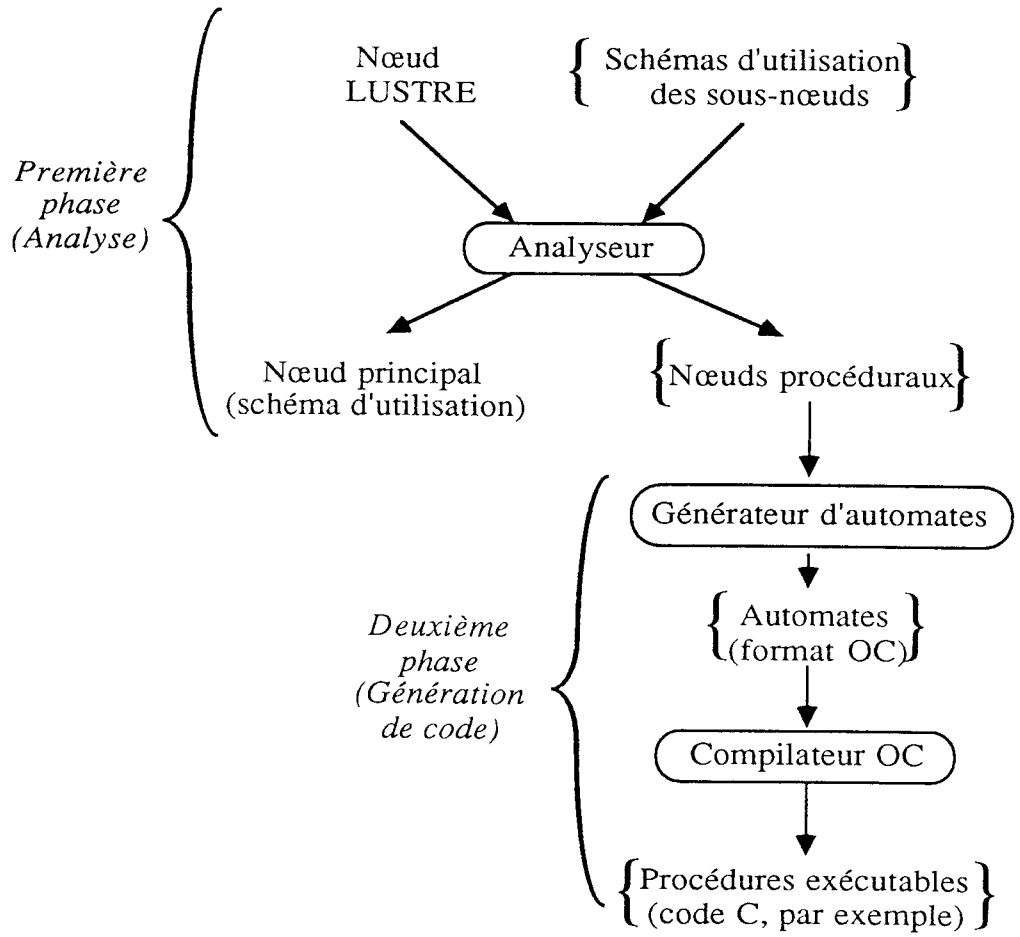


Fig.5.6. Architecture du compilateur LUSTRE.

Conclusion

L'objectif de ce travail était basé sur deux idées principales : ne pas remettre en cause (ou très peu) la génération d'automates, qui permet de produire pour les programmes LUSTRE du code simple et efficace ; étudier, avec cette restriction, les possibilités de compilation séparée dans le but de réduire le nombre d'états pour un programme.

La compilation séparée consiste en général à associer à toute abstraction procédurale un objet exécutable. Dans le cas des langages synchrones, si l'on veut procéder de cette manière, on se heurte au problème de la complexité de l'objet à générer. Pour le langage ESTEREL, G. Gonthier [4] avait mis en évidence ce problème. Il avait fait aussi remarquer qu'un programme ESTEREL "ne pouvant émettre aucune sortie avant que toutes les entrées soient disponibles", ne posait pas de problème de compilation séparée. Cette notion s'est donc tout à fait retrouvée dans le cas de LUSTRE avec les "nœuds procéduraux". Nous nous sommes donc dans un premier temps heurtés aux mêmes problèmes, pour arriver à une conclusion analogue : associer à tout nœud un objet exécutable unique ne constitue pas une solution acceptable, en ce sens que la complexité d'un tel objet ne répond pas au critère de fiabilité que l'on souhaite donner au code produit.

Cependant, bien que les problèmes soient les mêmes pour ces deux langages, dont la sémantique est très proche, l'aspect déclaratif du langage LUSTRE permet d'envisager des solutions basées sur la réécriture de programme, alors que les langages impératifs, comme ESTEREL, s'y prètent mal.

Nous avons donc mis à jour une méthode générale de réécriture de nœuds LUSTRE permettant de résoudre les problèmes de compilation. Il ne s'agit plus d'associer à tout nœud un objet exécutable, mais de restructurer ce nœud sous

une forme plus simple. Cette restructuration est paramétrée par :

- le choix d'un critère d'équivalence "data" ou "demand" (on retrouve donc la dualité propre à tout problème de compilation),
- le traitement particulier des variables temporelles, dont le rôle est primordial dans la synthèse du contrôle des programmes (variables d'état).

Pour ce qui est des perspectives, il nous faut conclure en émettant quelques réserves. La compilation séparée ne peut pas constituer une méthode générale. Son application systématique nous amènerait en effet à produire d'énormes réseaux de petits automates. Si l'on reprend l'exemple du nœud SOURIS (chapitre 2), on obtient déjà un réseau de trois automates à trois états, alors que la méthode traditionnelle d'expansion n'en produit qu'un (de plus très simple). Cette première réserve correspond à ce que nous avions dit dans le chapitre 3 : on ne réduit le nombre global d'états que si les nœuds du réseaux diffèrent de plus de deux variables de contrôle. Ces deux variables qui font la différence ne sont donc pas à négliger tant qu'on n'écrit que de petites applications. La compilation séparée doit donc rester un recours dans le cas de grosses applications, où le très grand nombre de variables de contrôle peut assurer à priori son succès.

Enfin, même dans le cas d'applications importantes, on peut préférer une solution à un seul automate, coûteuse en espace, à une solution réseau d'automates moins rapide. En effet la dispersion du controle induite par la compilation séparée va forcément augmenter le temps de réaction du programme.

Dans un premier temps, la compilation séparée sera donc une option proposée dans le cadre du compilateur LUSTRE-V3. A plus long terme, on peut envisager un compilateur capable de choisir lui-même la structure du programme produit, à partir de nouveaux paramètres comme l'emcombrement et la vitesse de réponse souhaités par le programmeur.

Références bibliographiques

[1] J.-L. Bergerand. *LUSTRE : un langage déclaratif pour le temps réel.* Thèse, Institut National Polytechnique de Grenoble, Grenoble, France, 1986.

[2] G. Berry, P. Couronné, G. Gonthier. Programmation synchrone des systèmes réactifs : le langage ESTEREL. *Technique et Science Informatique*, 4:305-316, 1987.

[3] P. Caspi, N. Halbwachs, D. Pilaud, J. A. Plaice. LUSTRE: a declarative language for programming synchronous systems. In *proc. 14th Annual ACM Symposium on Principles of Programming Languages*, pages 178-188, München, B.R.D., 1987.

[4] G. Gonthier. *Sémantiques et modèles d'exécution des langages réactifs synchrones; Application à ESTEREL.* Thèse, Université Paris VI, Paris, France, 1988.

[5] P. LeGuernic, A. Benveniste, P. Bournai, T. Gautier. Signal : a data flow oriented language for signal processing. *IEEE-ASSP*, 34(2):362-374, 1986.

[6] J. A. Plaice and J.-B. Saint. The LUSTRE-ESTEREL portable format. 1987. Non publié.

[7] J. A. Plaice. *Sémantique et compilation de LUSTRE, un langage déclaratif synchrone.* Thèse d'état, Institut National Polytechnique de Grenoble, Grenoble, France, 1988.

[8] W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language.* Academic Press, 1985.